intel®

Development Solutions

ICE™-5100 Emulator Reference Manual

ICE™-5100 Emulator Reference Manual

Order Number: 166257-003

Notational Conventions

The following notational conventions are used throughout this manual.

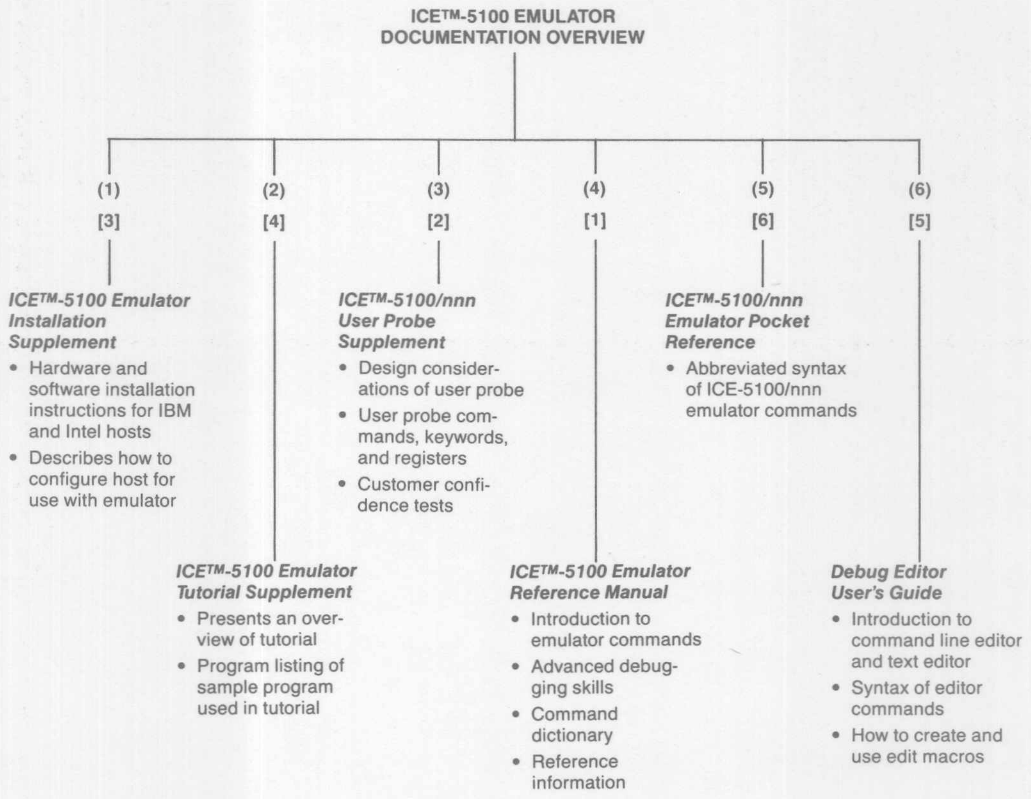
italics	indicate variable expressions. Substitute a value or symbol.
{*items*}	between braces indicate you must select one and only one item.
[*items*]	between brackets are optional items. You may select more than one item.
. . .	the preceding item may be repeated.
,. . .	the preceding item may be repeated with a comma between each repetition.
( )	are necessary to enclose an expression such as (#35 + 5).
device	stands for the number or letter of a disk drive.
dirname	stands for any user-created directory.
pathname:	is the fully-qualified reference to a file.
'	denotes your keyboard's apostrophe (') key. If your keyboard has two apostrophe keys, determine which one the ICE-5100 emulator accepts in command syntax.
CNTL	denotes the host keyboard's control key. For example, CNTL-C means enter C while pressing the control key.
<RETURN>	denotes the carriage return key.
`shading`	indicates user input.
punctuation	other than ellipses (. . .), braces ({ }), and brackets ([ ]) must be entered as shown.

Contents

166267-001

ICE™-5100 EMULATOR DOCUMENTATION OVERVIEW

The documentation for the ICE™-5100 emulator is divided into several manuals as shown in the following family tree. The number in parentheses indicates the recommended order of usage for the first-time user. The number in brackets is the recommended order of placement in the binder.



ICE™-5100 EMULATOR DOCUMENTATION OVERVIEW

(1) [3]

ICE™-5100 Emulator Installation Supplement

- Hardware and software installation instructions for IBM and Intel hosts
- Describes how to configure host for use with emulator

(2) [4]

ICE™-5100 Emulator Tutorial Supplement

- Presents an overview of tutorial
- Program listing of sample program used in tutorial

(3) [2]

ICE™-5100/nnn User Probe Supplement

- Design considerations of user probe
- User probe commands, keywords, and registers
- Customer confidence tests

(4) [1]

ICE™-5100 Emulator Reference Manual

- Introduction to emulator commands
- Advanced debugging skills
- Command dictionary
- Reference information

(5) [6]

ICE™-5100/nnn Emulator Pocket Reference

- Abbreviated syntax of ICE-5100/nnn emulator commands

(6) [5]

Debug Editor User's Guide

- Introduction to command line editor and text editor
- Syntax of editor commands
- How to create and use edit macros

ICE™-5100 EMULATOR REFERENCE MANUAL

Order Number: 166257-003

REV.	REVISION HISTORY	DATE
-001	Original issue.	8/86
-002	Update to correct errors.	9/86
-003	Final typeset version.	10/86

CAUTION

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instruction manual, may cause interference to radio communications. As temporarily permitted by regulation, it has not been tested for compliance with the limits for Class A Computing Devices pursuant to Subpart J of Part 15 of FCC rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Above	i$_m$	iSBC	PC BUBBLE
BITBUS	iMDDX	iSBX	Plug-A-Bubble
COMMputer	iMMX	iSDM	PROMPT
CREDIT	Insite	iSXM	Promware
Data Pipeline	Intel	KEPROM	QueX
FASTPATH	int$_e$l	Library Manager	QUEST
GENIUS	int$_e$lBOS	MAP-NET	Quick-Pulse Programming
$\overset{\Delta}{\text{i}}$	Intelevision	MCS	Ripplemode
i	int$_e$ligent Identifier	Megachassis	RMX/80
I^{2}ICE	int$_e$ligent Programming	MICROMAINFRAME	RUPI
ICE	Intellec	MULTIBUS	Seamless
iCEL	Intellink	MULTICHANNEL	SLD
iCS	iOSP	MULTIMODULE	UPI
iDBP	iPDS	ONCE	VLSiCEL
iDIS	iPSC	OpenNET	4-SITE
iLBX	iRMX	OTP	

CONTENTS

PREFACE

This manual is divided into the following chapters:

Chapter 1	presents an overview of the ICE™-5100 emulator software.
Chapter 2	presents debugging techniques and advanced ICE-5100 emulator features.
Chapter 3	is an encyclopedia of ICE-5100 emulator commands, keywords, and related topics.
Appendix A	describes the state of the ICE-5100 emulator when power is first turned on.
Appendix B	contains a list of miscellaneous topics you should be aware of when using the ICE-5100 emulator.
Appendix C	describes use of the clips assembly and the hardware specifications on the clips assembly.
Appendix D	contains hardware specifications on the power supply and serial cable pin-outs.
Appendix E	lists the error messages displayed by the ICE-5100 emulator.
Appendix F	lists ASCII codes and their functions.
Appendix G	lists related reference publications.
Glossary	contains a list of terms used in this manual.

Getting Started 1

166267-001

1 GETTING STARTED CONTENTS

intel

1 GETTING STARTED

1.1 Introduction

This chapter presents an overview of the ICE™-5100 emulator software. It gives you a simplified introduction to using the emulator's commands and features. Chapter 2 expands on the topics covered in this chapter as well as introducing new debugging techniques.

The examples in this chapter are based on the sample program (MESSG) used in the tutorial. If you so wish, you can load the sample program and follow along with the examples in this chapter; Section 1.11 explains how to load the program.

ICE-5100 emulator commands are shown in UPPERCASE; user defined variables are shown in lowercase. User input is `shaded` to distinguish it from responses returned by the emulator.

Refer to Chapter 3 for a complete description of ICE-5100 emulator commands and topics. Refer to your user probe supplement for information on user probe commands and registers.

1.2 ICE™-5100 Emulator Features

The ICE-5100 emulator is a high-level, interactive debugging system. It is used to speed up software and hardware development of selected MCS® -51 microcontroller components. With the ICE-5100 emulator you can:

- Execute ASM-51 and PL/M-51 programs.
- Map up to 64 KB of memory to the ICE-5100 emulator or user target system.
- Display and modify memory and registers.
- Create break registers to control program execution.
- Create trace registers to selectively collect trace information during emulation.
- Single-step through program instructions.
- Edit commands, debug definitions, and source files with an internal CRT-oriented text editor.
- Recall previous commands for execution.
- Obtain on-line help.
- Disassemble and assemble instructions.

1.3 Preparing a Program for ICE™-5100 Emulator Use

The ICE-5100 emulator provides symbolic debug support for programs written in the ASM-51 and PL/M-51 languages. Assemble or compile your program using the DEBUG option. The DEBUG option causes symbolic information to be included in the ASM-51 or PL/M-51 generated object file and allows symbolic use of program variables, labels, and line numbers during a debug session.

The following examples are typical ASM-51 and PL/M-51 compilation commands. The first example assembles an ASM-51 source file and the second example compiles a PL/M-51 source file.

```
ASM51 mod1.A51 DEBUG

PLM51 mod2.P51 DEBUG
```

In the preceding examples, the non-absolute object files mod1.obj and mod2.obj are generated by ASM-51 and PL/M-51 respectively.

After assembling (or compiling) your program modules, link the object files using the RL51 program. The RL51 program relocates and combines the non-absolute object files (with the PL/M-51 library routine file, if necessary) to produce a single executable absolute object file. Do not specify the NODEBUGLINES, NODEBUGSYMBOLS, or NODEBUGPUBLICS options when linking with RL51.

The following RL51 command links the files mod1.obj and mod2.obj with the PL/M-51 library module and produces an absolute object file named mod1.

```
RL51 mod1.obj, mod2.obj, plm51.lib
```

The RL51 output file may be specified with the "TO *filename*" option. Otherwise, the output filename will be the same as the first module in the link list with no extension.

The RL51-generated absolute object file can now be used with the ICE-5100 emulator via the LOAD command (refer to Section 1.11).

1.4 Invoking the ICE™-5100 Emulator Software

Invoke the ICE-5100 emulator software by entering the following command (*nnn* is the number of your user probe, and RUN is an ISIS utility program required for use with the Series III host):

```
[RUN] pathnameICEnnn
```

The emulator displays the following sign-on message followed by the prompt (hlt>) after successfully invoking the software:

```
host-operating-system-id ICE-5100/nnn Vx.y
Copyright 1986 INTEL CORPORATION
hlt>
```

1.5 ICE™-5100 Emulator Prompts

The ICE-5100 emulator uses different prompts in response to commands you enter. The following is a summary of the prompts:

hlt>	is the halt prompt. The ICE-5100 emulator is not in emulation. You can enter commands, interrogate program variables, or define debug object definitions.
emu>	is the emulation prompt. The ICE-5100 emulator is in emulation. You can enter any ICE-5100 emulator command not requiring probe access (refer to the GO entry in Chapter 3 for a list of invalid commands).
.hlt>	indicates a compound command. DO/END, REPEAT, COUNT and IF commands may be nested to any level depending on available memory. The following lines of a compound command get a_prompt preceded by a dot to indicate the command is inside a compound command. As commands are nested, the number of dots preceding the prompt increases. The following example illustrates command nesting:

```
hlt>DEFINE PROC ASK = DO
.hlt>WRITE 'Do you want to resume emulation?'
.hlt>DEFINE CHAR see_if = CI
.hlt>IF (see_if == 'y') or (see_if == 'Y') THEN
..hlt>RETURN FALSE
..hlt>ELSE RETURN TRUE
..hlt>ENDIF
.hlt>END
hlt>
```

Terminate compound commands with END to avoid syntax errors.

hlt>>	indicates an incomplete command. The double prompt indicates the emulator is waiting for you to complete the command. An example is typing EDIT FILE and pressing the carriage return key. The software continues prompting you until you enter the name of a file to edit or abort the command by typing <CNTL>C (<CNTL><BREAK> on IBM hosts).

1.6 Controlling Screen Display

The ICE-5100 emulator provides three paging commands for controlling the display of text on the screen:

P(age)	displays a page of information to the screen at one time. This is the initial default setting.
F(ast)	causes the information to scroll continuously to the screen.
L(ine)	causes the information to be displayed to the screen one line at a time.

Enter the first letter of a paging command *while* the information is scrolling to the screen.

1.7 Getting Help

The HELP command provides on-line assistance. To display a list of topics for which help is available, enter HELP as shown in Figure 1-1 (example is for the ICE-5100/252 user probe).

To get HELP on a specific topic, enter HELP *name-of-topic* as shown in the following command:

```
hlt> HELP EXIT
The EXIT command terminates a debug session.
hlt>
```

1.7.1 Error Information

Error messages followed by an asterisk enclosed within brackets [*] indicate extended information is available for that error. To display the extended message, enter HELP E*n*, where *n* is the number of the error message. For example, to get the extended message for error #95, enter:

```
hlt> HELP E95
#95
Invalid partition. [*]
A memory partition was specified in which the first address is
greater than the second address. For example, ASM 200H TO 100H
is an invalid partition.
```

1.8 Entering Commands

Emulator commands can be entered in either uppercase or lowercase letters. Enter the commands at the prompt (hlt> or emu>). The following sections describe the format for entering commands.

1.8.1 Extending a Command to Another Line

Commands that exceed 80 characters continue off the screen; the 80th character is displayed as an exclamation point (!). To extend a command to another line, enter an ampersand (&) followed by a carriage return and complete the command on the next line. Text that appears between the ampersand and the carriage return is interpreted as a comment.

The following example shows a command that extends over more than one line. Note that the ampersand causes the next line to have a double prompt.

```
hlt> GO FROM 0 USING break1 &
hlt>> TRACE trace1
emu>
```

```
hlt>HELP
HELP is available for:
        ADDRESS      APPEND       ASM          BASE         BIT          BOOLEAN
        BRKREG       BYTE         CHAR         CI           CNTL_C       COMMENTS
        CONSTRUCTS   COUNT        CPU          CURHOME      CURX         CURY
        DCI          DEBUG        DEFINE       DIR          DISPLAY      DO
        DYNASCOPE    EDIT         ERROR        EVAL         EXIT         EXPRESSION
        GO           HELP         IF           INCLUDE      INVOCATION   ISTEP
        KEYS         LABEL        LINES        LIST         LITERALLY    LOAD
        LSTEP        MAP          MENU         MODIFY       MODULE       MSPACE
        MTYPE        NAMESCOPE    OPERATOR     PAGING       PARTITION    PRINT
        PROC         PSEUDO_VAR   PUT          REFERENCE    REGS         REMOVE
        REPEAT       RESET        RETURN       SAVE         STRING       SYMBOLIC
        SYNCSTART    TEMPCHECK    TRCREG       TYPES        VARIABLE     VERIFY
        VERSION      WAIT         WORD         WRITE
```

Figure 1-1 HELP Menu

You may omit the ampersand if the command needs more syntactic information to be complete. If you enter a carriage return and the command is incomplete, the ICE-5100 emulator prompts for more information. For example:

```
hlt>EDIT FILE
hlt>>
```

1.8.2 Aborting Commands

Abort an ICE-5100 emulator command by entering <CNTL>C on Intel hosts and <CNTL><BREAK> on IBM hosts. <CNTL>C and <CNTL><BREAK> do not stop emulation; use the HALT command to stop emulation (see Section 1.14).

1.8.3 Entering Multiple Commands on a Line

To enter more than one command on a line, separate each command with a semicolon (;). The following example shows two commands on the same line.

```
hlt>CURHOME; CLEAREOS
hlt>
```

1.8.4 Retrieving and Re-Executing Previous Commands

Press <CNTL>E to re-execute the last command entered from the keyboard.

To recall a previously entered command for execution or editing, use the ↑ and ↓ keys to search through the emulator's 400 character history buffer for the desired command.

To retrieve a command from the history buffer, scroll through the buffer by pressing the ↑ key until you reach the command you want. Enter a carriage return to execute that command or use the command line editor keys listed in Table 1-1 to modify the command prior to executing.

The new version of the command becomes the latest entry in the command buffer. The old version is still in its original place in the buffer. Any changes made to a command using the editor (invoked by typing the EDIT command or pressing the <ESC> key) are not saved in the history buffer until after you execute them.

Refer to the *Debug Editor User's Guide*, order number 167098, for information on using the editor.

1.8.5 Creating Command Abbreviations

The LITERALLY command enables you create user defined character strings for use in debugging sessions. LITERALLYs are useful for creating command abbreviations for the ICE-5100 emulator.

NOTE

If you are familiar with using Intel's EMV-51 or ICE-51 emulators and their command syntax, use the LITERALLY command to create similar commands.

The following LITERALLY definitions correspond to ICE-5100 emulator commands.

```
DEFINE LITERALLY def = 'DEFINE'
DEFINE LITERALLY lit = 'LITERALLY'
DEF LIT bx = 'BYTE XDATA'
DEF LIT g = 'GO'
DEF LIT g0 = 'GO FROM 0'
DEF LIT h = 'HALT'
DEF LIT rc = 'RESET CHIP'
DEF LIT pa = 'PRINT ALL'
DEF LIT brk = 'DEFINE BRKREG'
DEF LIT trc = 'DEFINE TRCREG'
DEF LIT inc = 'INCLUDE'
```

Typing in the LITERALLY name followed by a space expands the LITERALLY definition to its true meaning. For example, assuming you had defined the preceding LITERALLYs, typing g followed by a space would cause the ICE-5100 emulator to expand the letter g to its LITERALLY meaning of GO.

LITERALLY definitions can be placed inside of an ICE-5100 emulator macro file. Then each time you invoke the emulator software, the definitions are included. Refer to Chapter 2 and the ICE*nnn* entry in Chapter 3 for information on creating an ICE*nnn*.MAC file.

The tutorial includes a sample macro file for your use.

Table 1-1 Command Line Editor Keys

Key Name	Function
<RUBOUT>	Deletes the character to the left of the cursor.
<CNTL>A	Deletes the line to the right of the cursor.
<CNTL>C	Cancels the command in progress. Use CNTL-BREAK on IBM hosts.
<CNTL>E	Re-executes the last command.
<CNTL>F	Deletes the character at the cursor.
<CNTL>X	Deletes the line to the left of the cursor.
<CNTL>Z	Deletes the current line.
<ESC>	Invokes the text editor and places the present command in the edit buffer for editing. If you press <ESC> at the prompt, the previous command is retrieved from the history buffer for editing.
<←>	Moves the cursor left one character.
<→>	Moves the cursor right one character.
<↑>	Retrieves the previous line from the history buffer.
<↓>	Moves to the next line in the history buffer.
<HOME>	Magnifies the effect of the preceding arrow key. Causes jumps to the beginning or end of the current line when used with the right or left arrow.

1.8.6 Commenting Commands

Enclose comments within a slash-asterisk (/* comment */) combination. A /* begins the comment, and an */ ends the comment. The following example illustrates using a comment to document a command.

```
hlt> GO TIL rotate      /* Stop at start of ROTATE procedure */
hlt>
```

1.9 Using the Syntax Guide

The integrated command directory (menu) is a syntax guide at the bottom of the screen. It lists your options when entering emulator commands. Watch the menu as you enter commands; it changes to indicate command options available. If you follow the choices, you cannot construct a syntactically incorrect command, although it may be semantically incorrect.

The following menu is the first menu you see after invoking the software (example is based on the 252 user probe):

```
---- more ---- Use <TAB> to cycle through prompts.
APPEND ASM BASE BRKREG CAUSE CLEAREOL CLEAREOS COUNT CPU
```

In many cases, the complete menu will not fit on one line. Circle through the menu by pressing the TAB key.

```
---- more ----
CURHOME DEFINE DIR DO EDIT EVAL EXIT <expr> GO HALT HELP
```

Use the MENU command to control the menu display. Enter MENU = FALSE to suppress menu display and MENU = TRUE to display the menu. <CNTL>V toggles the menu on and off.

Figure 1-2 shows how the syntax menu changes for the GO command. Note the semi-colon option. It is a command delimiter and indicates a position where you can enter another command on the same line.

(Example assumes br1 and trc1 have been defined.)

1.10 Mapping Memory

You need to map program memory to ICE memory before loading a program for debugging. Use the MAP command to display or modify the 64-KB mappable memory. Enter MAP by itself to display the current MAP setting (the following example is based on the ICE-5100/252 user probe):

```
hlt> MAP
MAP        0K  LENGTH    8K  ICE
MAP        8K  LENGTH   56K  USER
[USER EA PIN = 1]
hlt>
```

On power-up all memory is mapped to USER (except for on-chip ROM when EA is not asserted).

To map memory to ICE, enter the MAP command and indicate the amount of memory to be mapped in 4 KB blocks starting on 4KB boundaries. For example, to map 12 KB of memory to ICE, enter (*n* is the upper boundary of your user probes's on-chip memory, e.g., 8K for the ICE-5100/252 user probe):

```
hlt> MAP nK LENGTH 12K ICE
hlt>
```

To map 64 KB of memory to ICE, enter:

```
hlt> MAP ICE
hlt>
```

1.11 Loading Programs

After mapping memory to ICE, use the LOAD command to load a program for execution. The ICE-5100 emulator accepts only MCS-51 absolute object files or hex-format object files.

```
hlt> GO

FROM ARM FOREVER  TIL USING  TRACE   ;   <execute>

hlt> GO FROM

<expr>

hlt> GO FROM 13H

<operator>   ARM FOREVER  TIL USING  TRACE   ;   <execute>

hlt> GO FROM 13H USING

BRKREG   <brkreg name>

hlt> GO FROM 13H USING br1

, TRACE  ;  <execute>

hlt> GO FROM 13H USING br1 TRACE

<expr> OUTSIDE PAGE FROM TIL <trcreg name>  ;  <execute>

hlt> GO FROM 13H USING br1 TRACE trc1

; <execute>
```

Figure 1-2 The Integrated Command Directory for the GO Command

The following LOAD command loads the sample program supplied with the ICE-5100 emulator tutorial. If you want to load the program and follow along with the examples in this chapter and Chapter 2, enter:

```
hlt> LOAD messg
hlt>
```

1.12 Displaying Program Symbols and Debug Objects

Use the DIR command to display user-defined debug objects and program symbols.

The DIR command by itself displays the directory for the current module.

```
hlt> DIR
DIR of :MAIN_DISPLAY
   RESET_LOW  . . label
   RESET_HIGH   . label
   DISP_BUFFER . <byte>
   BUFF_SIZE . . <byte>
   INT_FLAG  . . bit
   MESSAGE . . . label
   TMO_LOW . . . <byte>
   TMO_HIGH  . . <byte>
   INIT  . . . . procedure
      I  . . . . . . <byte>
      TMOD . . . . . <byte>
      ETO  . . . . . bit
      EA . . . . . . bit
      TRO  . . . . . bit
   SERVICE . . . procedure
   CHAR_DISPLAY procedure
   ROTATE  . . . procedure
      INDEX_PTR  . . <byte>
      TEMP . . . . . <byte>
   START  . . . . label
hlt>
```

1.12.1 Directory of Program Lines

DIR LINE displays a directory of line numbers. Line numbers are not available in ASM-51 programs.

```
hlt> DIR LINE
DIR of :MAIN_DISPLAY
#1   #3   #5   #6   #7   #8   #9   #10  #11  #12
#13  #14  #15  #16  #17  #18  #19  #20  #21  #22
#23  #24  #26  #27  #28  #29  #30  #31  #32  #33
#34  #35  #36  #37  #38  #39  #40  #41
hlt>
```

1.12.2 Directory of Program Procedures

DIR PROCEDURES displays a directory of program procedures.

```
hlt> DIR PROCEDURE
DIR of :MAIN_DISPLAY
SERVICE
CHAR_DISPLAY
ROTATE
hlt>
```

1.12.3 Directory of Program Symbols

DIR SYMBOLS displays a list of program symbols (example only shows a partial listing of symbols for MAIN_DISPLAY).

```
hlt> DIR SYMBOLS
DIR of :MAIN_DISPLAY
RESET_LOW . . label
RESET_HIGH . label
DISP_BUFFER . <byte>
BUFF_SIZE . . <byte>
INT_FLAG  . . bit
MESSAGE . . . label
hlt>
```

1.12.4 Directory of Program Modules

DIR MODULE displays a list of program modules.

```
hlt> DIR MODULE
MAIN_DISPLAY
hlt>
```

1.13 Beginning Program Execution

Enter the GO command to begin program execution at the current execution point or at a specified address. Program execution is indicated by the yellow LED on the ICE-5100 controller pod and by the prompt:

```
emu>
```

GO FOREVER is the initial default condition. GO FOREVER causes program execution to proceed without any break specifications. Executing the GO command by itself starts program execution from the current execution point and with the last GO specifications.

The following example initially executes a program forever.

```
hlt> GO
emu>
```

1.14 Stopping Program Execution

Stop program execution by using the HALT command.

```
emu> HALT
Probe stopped at 008CH (:MAIN_DISPLAY) because of user halt.
hlt>
```

1.15 Using GO Command Options

The following example forces execution to begin at location 0 and executes MESSG until memory location 24 is reached.

```
hlt> GO FROM 0 TIL 24H
emu>
Probe stopped at 0026H (:MAIN_DISPLAY#7 + BH) because of internal break.
hlt>
```

The following example executes MESSG until line number 24 is reached.

```
hlt> GO TIL #24
emu>
Probe stopped at 0061H (:MAIN_DISPLAY#27) because of internal break.
hlt>
```

The following example uses a symbolic reference to halt emulation at the start of the ROTATE procedure.

```
hlt> GO TIL :main_display.rotate
emu>
Probe stopped at 0061H (:MAIN_DISPLAY#27) because of internal break.
hlt>
```

GO FOREVER executes MESSG forever.

```
hlt> GO FOREVER
hlt>
```

Chapter 2 contains more advanced options for use with the GO command.

1.16 Stepping Through Programs

The ICE-5100 emulator has two commands that can be used to step through programs.

Use the **ISTEP** command to single-step through user programs by machine language instructions. This can be done one instruction at a time or a specified number of instructions. ISTEP displays the current execution point ($) and an opcode disassembly of the next instruction when execution halts.

Step through the instruction at address 24H.

```
hlt> ISTEP FROM 24H
     0026H F8          MOV      R0,A
hlt>
```

Step through 10 instructions beginning at address 24H.

```
hlt> ISTEP 10 FROM 24H
     0022H E508        MOV      A,I
hlt>
```

Use the **LSTEP** command to single-step through user programs by program line numbers. This can be done one line at a time or a specified number of lines. LSTEP displays the next high-level statement number when execution halts. LSTEP requires high-level language statement information. Therefore LSTEP does not work with ASM-51 programs.

LSTEP through line #5 of MAIN_DISPLAY.

```
hlt> LSTEP FROM :main_display#5
:MAIN_DISPLAY#6
hlt>
```

LSTEP through a range of program lines starting from 0EH.

```
hlt> LSTEP 5 FROM 0EH
:MAIN_DISPLAY#8
hlt>
```

1.17 Assembling and Disassembling Program Code

Use the ASM command to display or modify program code memory as ASM-51 assembly mnemonics. The syntax determines if memory is to be displayed or modified. If the assignment portion of the command is not present, memory is displayed. When the assignment portion is present, the assembly mnemonic is translated into object code and placed in memory at the specified address.

To disassemble program memory, enter the ASM command and an address. In the following example the address is the one stored in the program counter, referenced by $.

```
hlt> ASM $
     ADDR  CODE        INSTRUCTION
     0024H 2421        ADD     A,#21H      ; +033T
hlt>
```

To disassemble a range of instructions (in this example, five instructions starting from the one pointed to by $), enter:

```
hlt> ASM $ LENGTH 5
     ADDR  CODE          INSTRUCTION
(:MAIN_DISPLAY#8)
     0029H 7808          MOV     R0,#08H        ;+008T
     002BH 7401          MOV     A,#01H         ;+001T
     002DH 26            ADD     RA,@R0
     002EH F6            MOV     @R0,A
     002FH 50E3          JNC     0014H          ;$-1BH
hlt>
```

There are **two methods for assembling** ASM-51 mnemonics into memory.

The first method requires entering the ASM command with a starting address followed by an equal sign and each assembly mnemonic enclosed within single-quotes ('). Separate mnemonics with commas.

The following example shows assembling two instructions into memory starting at 04B5H:

```
hlt> ASM 04B5H = 'mov r0, #20', 'mov a, @r1'
hlt>
```

You can also **use the ASM line mode to assemble mnemonics into memory.** This form of assembling mnemonics is useful when you need to know where you are assembling (e.g., when creating program patches).

Enter the ASM command and the address where you want to begin assembling followed by an equal sign and a carriage return. The specified address appears on the screen as a prompt. Enter one assembly mnemonic per line (do not enclose the mnemonic in quotes when using line mode). The address prompt is updated following each carriage return. Terminate the assembly with a carriage return on a blank line.

The following example shows using ASM line mode beginning at 04B5H:

```
hlt> ASM 04B5 = <RETURN>
>>>>>>04B5> Assembler Line Mode: enter blank line to terminate. <<<<<<
04B5H> mov r0, #20
04B7H> inc r1
04B8> <RETURN>
hlt>
```

1.18 Manipulating Program Objects

The following sections describe commands and methods used to reference and modify memory locations and registers.

1.18.1 Referencing Memory Locations

Use a colon (:) to reference the address of a program module (main_display).

```
hlt> :main_display
CODE    000EH
hlt>
```

Symbolically display the value of a program variable (disp_buffer) by entering the variable name.

```
hlt> disp_buffer
7E
hlt>
```

Symbolically change the value of a program variable (disp_buffer) by entering the variable name followed by an equal sign and the new value.

```
hlt> disp_buffer = 15
hlt>
```

Place a dot (.) before a program variable name to return the address rather than the value. For example:

```
hlt> .disp_buffer
IDATA   0021H
hlt>
```

Use the double-quote operator (") to reference program variables with the same name as an ICE-5100 emulator keyword. The following example shows using the double-quote operator before the label print (ICE-5100 emulator keyword):

```
hlt> .:main_display.rotate."print
CODE 0085H
hlt>
```

When the variable you wish to change is not a BIT or BYTE *mtype* (memory type), you must use an *mtype* command as shown in the following example (disp_buffer is a BYTE array):

```
hlt> BYTE .disp_buffer LENGTH 50H = 0
hlt>
```

1.18.2 Qualifying References

Use the NAMESCOPE pseudo-variable as a reference point to determine the qualification needed to symbolically reference a program variable. By default, whenever you load a program NAMESCOPE is set to the prologue of the main program module.

To display the current reference address, enter NAMESCOPE by itself.

```
hlt> NAMESCOPE
CODE    008AH
hlt>
```

Symbolic references to a program variable must be fully qualified unless the variable is within the current module or procedure designated by NAMESCOPE. A fully-qualified reference to a symbol includes the module name and the names of all procedures that enclose the symbol. For example, to reference the variable temp, enter:

```
hlt> :main_display.rotate.temp
74
hlt>
```

Changing NAMESCOPE affects the amount of qualification needed to reference a variable. The following example changes NAMESCOPE and allows partially-qualified references to program variables within the procedure rotate.

```
hlt> NAMESCOPE = :main_display.rotate
hlt>
```

Access the variable temp.

```
hlt> temp = 16
hlt> temp
16
hlt>
```

Refer to Chapter 2 for information on dynamically changing NAMESCOPE.

1.18.3 Evaluating Expressions

Use the EVAL LINE command to display the line number corresponding to memory address 24H.

```
hlt> EVAL 024H LINE
:MAIN_DISPLAY#7H + 9H
hlt>
```

EVAL PROCEDURE displays the name of the procedure corresponding to memory address 005EH.

```
hlt> EVAL 005EH PROCEDURE
:MAIN_DISPLAY.ROTATE
hlt>
```

Use the EVAL SYMBOL command to display memory location 20H as a program symbol.

```
hlt> EVAL 20H SYMBOL
(BIT   )  :MAIN_DISPLAY.INT_FLAG + 20H
(CODE  )  <unknown>
(IDATA)   :MAIN_DISPLAY.ROTATE.TEMP + 17H
(RDATA)   <unknown>
(XDATA)   <unknown>
hlt>
```

Use the EVAL command to display numbers or calculations in hexadecimal, decimal, binary, and ASCII.

```
hlt> EVAL 357 + 33H
0000001110001010 1Y   906T   038AH   '. .'
hlt>
```

1.19 Displaying Register Values

Use the REGS command to display the contents of selected microcontroller registers. For example (display is for the ICE-5100/252 user probe):

```
hlt> REGS
$      =0026H     TM0     =FDF9H     R0   =   02H     R4   =  0CH
ACC    =  0AH     TM1     =0000H     R1   =   FEH     R5   =  00H
SP     =  07H     TM2     =0050H     R2   =   06H     R6   =  03H
DPTR =10C6H      RBANK  =   00H     R3   =   B0H     R7   =  CAH
FLAGS:  CY=1      AC=0    F0=0    RS1=0    RS0=0    OV=0    P=0
hlt>
```

The RBANK command is used to display or select one of the general purpose register banks. For example, to select bank 2, enter:

```
hlt> RBANK = 2
hlt>
```

To display the contents of a special function register (SFR), specify just the SFR name. For example, to display the contents of the PSW register, enter:

```
hlt> PSW
10
hlt>
```

Change the contents of an SFR.

```
hlt>PSW = 12
hlt>
```

Display the address of an SFR by entering the SFR name preceded by a dot (.).

```
hlt> .PSW
RDATA 00D0H
hlt>
```

To display the contents of an SFR bit, enter the keyword corresponding to the desired bit. For example, to display the PSW parity flag, enter:

```
hlt> P
1
hlt>
```

To display the address of the PSW parity flag, enter:

```
    hlt>.PSW.0
    BIT 00D0H
    hlt>
```

or

```
    hlt>.P
    BIT 00D0H
    hlt>
```

Use the BIT keyword to change the contents of a bit memory location.

```
    hlt> BIT 00D0H = BYTE CODE :main_display.rotate.temp
    hlt>
```

Only the least significant bit of an expression or *mtype* is assigned to a bit register.

Refer to your user probe supplement for a list of special function register and bits.

1.20 Resetting the ICE™-5100 Emulator Software

In general, a RESET ICE command may be needed whenever a probe interface or serial communications error occurs. Reset the emulation processor to its power-up state with RESET CHIP.

You do not need to reload your programs after executing a RESET command.

Executing a RESET ICE command restores the ICE-5100 controller pod to its power-up state (refer to Appendix A for power-up conditions) and reloads the emulator software.

The following example executes a RESET ICE command and resets the baud rate to 19200.

```
    hlt> RESET ICE BAUD (19200)
    hlt>
```

Executing a RESET CHIP command resets the emulation processor, the interrupt-in-progress flag, and the stack pointer to their default states. In addition, ports P0, P1, P2, and P3 are reset to 0FFH, and all other on-chip registers are reset to 00H.

To reset the emulation processor, enter:

```
    hlt> RESET CHIP
    hlt>
```

1.21 Exiting the ICE™-5100 Emulator Software

Enter the EXIT command to terminate the debug session and return control to the host operating system. On exit, the ICE-5100 emulator closes all open files. The state of the ICE-5100 emulator hardware is not changed by the exit. The following example shows the EXIT command:

```
hlt> EXIT
ICE-5100/nnn terminated
```

Advanced
Emulation Topics 2

166267-001

2 ADVANCED EMULATOR TOPICS CONTENTS

Page

2 ADVANCED EMULATOR TOPICS

2.1 Introduction

This chapter expands on the topics discussed in Chapter 1. In this chapter you will be introduced to more advanced features of the ICE-5100 emulator and shown how to use them.

The examples used in this chapter are based on the sample program used in the ICE-5100 tutorial. If you want to load the program and follow along with the examples in this chapter, enter the following command:

```
hlt>LOAD MESSG
```

ICE-5100 emulator commands are shown in UPPERCASE; user defined variables are shown in lowercase. User input is shaded to distinguish it from responses returned by the emulator.

Refer to Chapter 3 for a complete description of ICE-5100 emulator commands and topics.

2.2 Differences in Debugging ASM-51 and PL/M-51 Programs

All of the examples used in the ICE-5100 emulator documentation are based on the PL/M-51 program used in the tutorial. PL/M-51 was used for readability and to demonstrate the full symbolic capability of the ICE-5100 emulator.

Debugging of ASM-51 programs is fully supported by the ICE-5100 emulator. The only difference between ASM-51 and PL/M-51 program support is that in ASM-51 you cannot reference line numbers. When debugging in ASM-51, use memory addresses or symbolic references as program references.

2.2.1 Program Listings

Figure 2-1 shows the PL/M-51 program listing for the ROTATE procedure while Figure 2-2 shows the ASM-51 program listing. Listings are presented in both languages to illustrate the available symbolic information in each.

Refer to Appendix B for further considerations which apply to both ASM-51 and PL/M-51 program support.

```
24  2  ROTATE: PROCEDURE;

25  2    DECLARE   INDEX_PTR   BYTE,
                   TEMP        BYTE;

         /* Begin subroutine processing */

26  2    INDEX_PTR = 1;
27  2    TEMP      = disp_buffer(0);

28  3    DO WHILE (index_ptr < buff_size);

29  3      disp_buffer(index_ptr - 1) =_= disp buffer(index_ptr);
30  3      INDEX PTR = INDEX_PTR + 1;

31  3    END;

32  2    disp_buffer(buff_size) = TEMP;

33  2  PRINT: CALL CHAR_DISPLAY;

34  2    int_flag = false;

35  2  END;
```

Figure 2-1 PL/M-51 Listing for the Rotate Procedure

2.3 Using an ICE*nnn*.MAC File

Macro files are used to store debug object defintions and ICE-5100 emulator commands. Create a macro file with any text editor (use the built-in debug editor). If you create a macro file and want it to be included each time you invoke the software, name the file ICE*nnn*.MAC (where *nnn* is the number of your user probe) and copy it to the *pathname* containing the ICE*nnn*.86 (or ICE*nnn*.EXE for IBM hosts) software file.

If you give the macro file a name other than ICE*nnn*.MAC and you want to use it during a debug session, there are two options available for including the file:

1. Specify MACRO and the macro file name when you invoke the software. For example:

```
   C:\ICEDIR>ICEnnn MACRO (useful.mac)
```

```
                         ; PROCEDURE ROTATE (START)
                                           ; STATEMENT # 26
 005E  750801  F             MOV   INDEX_PTR,#01H
                                           ; STATEMENT # 27
 0061  852109  F             MOV   TEMP,DISP_ BUFFER
                                           ; STATEMENT # 28
 0064                WHILE?5:
 0064  E508    F             MOV   A,INDEX_PTR
 0066  C3                    CLR   C
 0067  9553    F             SUBB  A,BUFF_SIZE
 0069  5013                  JNC   WEND?6
                                           ; STATEMENT # 29
 006B  E508    F             MOV   A,INDEX_PTR
 006D  2421    F             ADD   A,#DISP_BUFFER
 006F  F8                    MOV   R0,A
 0070  E508    F             MOV   A,INDEX_PTR
 0072  14                    DEC   A
 0073  8606                  MOV   AR6,@R0
 0075  2421    F             ADD   A,#DISP_BUFFER
 0077  F8                    MOV   R0,A
 0078  A606                  MOV   @R0,AR6
                                           ; STATEMENT # 30
 007A  0508    F             INC   INDEX_PTR
                                           ; STATEMENT # 31

 007C  80E6                  SJMP  WHILE?5
 007E                WEND?6:
                                           ; STATEMENT # 32
 007E  E553    F             MOV   A,BUFF_SIZE
 0080  2421    F             ADD   A,#DISP_BUFFER
 0082  F8                    MOV   R0,A
 0083  A609    F             MOV   @R0,TEMP
                                           ; STATEMENT # 33
 0085                PRINT:
 0085  115D    F             ACALL CHAR_DISPLAY
                                           ; STATEMENT # 34

 0087  C200    F             CLR   INT_FLAG
                                           ; STATEMENT # 35

 0089  22                    RET
                          ; PROCEDURE ROTATE (END)
```

Figure 2-2 ASM-51 Listing For The Rotate Procedure

2. Retrieve the macro file after invocation with the INCLUDE command. For example:

   ```
   hlt> INCLUDE C:\ICEDIR\useful.mac
   ```

   To suppress the display of the macro file on the screen as it is included, use the NOLIST option. For example:

   ```
   hlt> INCLUDE C:\ICEDIR\useful.mac NOLIST
   ```

   The following is a typical ICE*nnn*.MAC macro file:

```
BASE = HEX                                         /*Sets number base*/
MAP ICE                                    /*Maps 64 KB memory to ICE*/
LOAD messg                            /*Load a program for execution*/
DEFINE PROC crt = DO                    /*Create a debug procedure*/
CURY = 20T
CHAR .disp_buffer LENGTH buff_size
RETURN FALSE
END
DEFINE BRKREG crtbrk = rotate CALL crt              /*Create BRKREGs*/
DEFINE BRKREG brl = #24
DEFINE LITERALLY pa = 'PRINT ALL'  /*Create command abbreviations*/
DEFINE LITERALLY gu = 'GO USING'
DEFINE LITERALLY h = 'HALT'
INCLUDE sample.mac NOLIST                /*Retrieve another macro file*/
```

2.4 Making NAMESCOPE Dynamic

Use the DYNASCOPE command to enable NAMESCOPE to change whenever the execution point changes. Setting DYNASCOPE to TRUE causes the emulator to update NAMESCOPE to the current execution point following an execution break, $, ISTEP, LSTEP, or HALT command. When DYNASCOPE is FALSE, NAMESCOPE changes only if you load a program or assign NAMESCOPE a new address.

The following example illustrates the effect DYNASCOPE has on NAMESCOPE.

```
hlt> NAMESCOPE
CODE  008AH
hlt> DYNASCOPE = TRUE
hlt> GO TIL #24
emu>
Probe stopped at 0061H (:MAIN_DISPLAY#27) because of internal break.
hlt> NAMESCOPE
CODE  0061H
hlt>
```

Note that NAMESCOPE is now set to the current execution point. Allowing NAMESCOPE to dynamically change allows you to reference symbols within the current program module without fully qualifying each symbol with the name of the module containing the execution point.

2.5 Controlling Emulation

This section describes using the GO command options and debug objects to control emulation.

2.5.1 Using an ARM Condition to Break Emulation

The ARM option is used to conditionally enable breakpoints. Once an ARM address is executed, all specified breakpoints are enabled. ARM can be used with individual breakpoints or with BRKREGs.

The following example halts emulation at line #14 only after line #13 has been executed.

```
hlt> GO FROM 0 ARM #13 TIL #14
emu>
Probe stopped at 0048H (:MAIN_DISPLAY#15) because of internal break.
hlt>
```

2.5.2 Types of Breakpoints Allowed

The ICE-5100 emulator supports three types of breakpoints as described below:

- Specific break — Emulation halts when a specific instruction address is executed.
- Range break — Emulation halts when an instruction address within a specified range of addresses is executed. A range break counts as three specific breaks.
- Page break — Emulation halts when an instruction address within a page (256 bytes) is executed. Any number of page breaks count as one specific break.

The ICE-5100 emulator keeps track of the break types used and issues an error message if a GO command is entered with more breakpoints than the hardware can support. Up to four specific breaks (or equivalent) per GO command are accepted by the ICE-5100 emulator.

2.5.3 Using BRKREGs

A BRKREG contains one or more break specifications which are used in a GO command. Optionally, a BRKREG may call a debug procedure (PROC, refer to Section 2.5.4).

2.5.3.1 Creating BRKREGs

BRKREGs are created with the DEFINE command and may optionally be contained within a DO/END block.

The following BRKREG uses a memory address as a breakpoint.

```
    hlt>DEFINE BRKREG brk1 = 24H
    hlt>
```

The following BRKREG uses a program line number as a breakpoint.

```
    hlt>DEFINE BRKREG brk2 = #33
    hlt>
```

The following BRKREG uses a symbolic reference as a breakpoint.

```
    hlt>DEFINE BRKREG crt = :main_display.rotate
    hlt>
```

The following BRKREG contains multiple breakpoints. It is enclosed within a DO/END block. However, it would still be correct if the DO/END block was omitted.

```
    hlt>DEFINE BRKREG brk3 = DO #24, #33, #38 END
    hlt>
```

2.5.3.2 Executing BRKREGs

Execute a BRKREG with the GO USING command. You have the option of specifying individual BRKREGs by name or including all currently defined BRKREGs by using the BRKREG keyword.

Begin emulation from the start of the program using one BRKREG.

```
hlt> GO FROM 0 USING brk1
emu>
Probe stopped at 0026H (:MAIN_DISPLAY#7) +11T) because of internal break.
 Break Register is [BRK1]
hlt>
```

Begin emulation from the current execution point and use two BRKREGs to control emulation. Emulation is halted when any breakpoint in either BRKREG is reached.

```
hlt> GO USING brk1, crtbrk
emu>
Probe Stopped at 0061H (:MAIN_DISPLAY#27) because of internal break.
 Break Register is [CRTBRK]
hlt>
```

Begin emulation from the current execution point using all currently defined BRKREGs.

```
hlt> GO USING BRKREG
emu>
Probe stopped at 0026H (:MAIN_DISPLAY#7 + 11T) because of internal break.
 Break Register is [BRK]
hlt>
```

2.5.3.3 Removing Breakpoints

When you specify a breakpoint (using an address or BRKREG), the ICE-5100 emulator sets a hardware break at the location specified. From that point on, the breakpoint remains set until you execute a GO USING, GO TIL, GO FOREVER, or a RESET ICE command. For example:

```
hlt> GO FROM 0 USING brk1
emu>
Probe stopped at 0026H (:MAIN_DISPLAY #7 @ 11T) because of internal break.
 Break Register is [brk1]
hlt> GO
emu>
Probe stopped at 0026H (:MAIN_DISPLAY#7 + 11T) because of internal break
 Break Register is [BRK1]
hlt>
```

Clear breakpoints from memory.

```
hlt> GO FOREVER
emu>
```

or

```
hlt> GO USING brk2
emu>
Probe stopped at 0061H (:MAIN_DISPLAY#27) because of internal break
 Break Register is [BRK2]
hlt>
```

or

```
hlt> RESET ICE
hlt>
```

Refer to the GO and BRKREG entries in Chapter 3 for specifics on creating and using BRKREGs.

2.5.4 Using Debug Procedures

Debug procedures (PROCs) are used to collectively group and execute ICE-5100 emulator commands. The commands are contained within a DO/END block and are executed in the order they appear within the PROC.

Execute a PROC by entering its name. This can be done at the emulator prompt, from within a macro file, or from within a BRKREG.

Procs are useful in that they enable you to automate the software testing process. PROCs can be used to generate test values, simulate I/O hardware, selectively halt and resume emulation, or execute temporary program patches.

2.5.4.1 PROC Examples

The following PROC displays the status of ICE-5100 pseudo-variables. The PROC is part of the ICE-5100 emulator tutorial, and is included on the tutorial software disk in the file SAMPLE.MAC.

Refer to the WRITE entry in Chapter 3 for more information on the WRITE command's syntax.

```
hlt>DEFINE PROC status = DO
.hlt>DEFINE CHAR basestr = 'HEX'           /* Determine current number base */
.hlt>IF BASE == 10Y THEN
..hlt>basestr = 'BINARY'
..hlt>ENDIF
.hlt>IF BASE == 10t THEN
..hlt>basestr = 'DECIMAL'
..hlt>ENDIF
.hlt>                                        /* Display status information */
.hlt>WRITE USING &                                /* Set up display format */
.hlt>>('6C,"DYNASCOPE = ",0,30C,"ERROR = ",0,54C,"SYMBOLIC = ",0')&
.hlt>>DYNASCOPE, ERROR, SYMBOLIC
.hlt>WRITE USING &
.hlt>>('6C,"SYNCSTART = ",0,30C,"TEMPCHECK = ",0,54C,"VERIFY = ",0')&
.hlt>>SYNCSTART, TEMPCHECK, VERIFY
.hlt>WRITE USING &
.hlt>>('6C,"BASE = ",0,30C,"RBANK = ",0') basestr, RBANK
.hlt>END
.hlt>
```

Execute the PROC by entering its name.

```
hlt>status
    DYNASCOPE =  FALSE      ERROR     =  TRUE      SYMBOLIC =  TRUE
    SYNCSTART =  FALSE      TEMPCHECK =  TRUE      VERIFY   =  TRUE
    BASE      =  DECIMAL    RBANK     =  02
hlt>
```

The following PROC displays the IE register in binary form with headings corresponding to each bit. Note the usage of the NUMTOSTR and SUBSTR commands. NUMTOSTR converts a number to a character string and SUBSTR extracts a specified protion of the character string.

```
hlt>DEFINE PROC iestat = DO   /* Display bits of IE register with headings */
.hlt>WRITE ' '
.hlt>WRITE '                      IE REGISTER (Interrupt Enable)'
.hlt>WRITE ' '
.hlt>WRITE &                                    /* Continue on next line */
.hlt>>'ENABLE  PCA   Timer2  Serial  Timer1  Extrn1  Timer0  Extrn0'
.hlt>WRITE &
.hlt>>' (EA)   (EC)   (ET2)   (ES)   (ET1)   (EX1)  (ET0)   (EX0)'
.hlt>WRITE &
.hlt>>'---------------------------------------------------------'
.hlt>BASE = BINARY                          /* Set number base to BINARY */
.hlt>DEFINE CHAR tmp = NUMTOSTR (IE)  /* Convert BINARY to character string */
.hlt>WRITE ' ', &
.hlt>>SUBSTR (tmp,1t,1),'   ',&      /* Strip off individual bits in string */
.hlt>>SUBSTR (tmp,2t,1),'   ',&                        /* Write out bits */
.hlt>>SUBSTR (tmp,3t,1),'   ',&
.hlt>>SUBSTR (tmp,4t,1),'   ',&
.hlt>>SUBSTR (tmp,5t,1),'   ',&
.hlt>>SUBSTR (tmp,6t,1),'   ',&
.hlt>>SUBSTR (tmp,7t,1),'   ',&
.hlt>>SUBSTR (tmp,8t,1),'   ',&
.hlt>WRITE ' '
.hlt>BASE = HEX
.hlt>END
hlt>
```

Execute the PROC.

```
hlt>iestat
                      IE REGISTER (Interrupt Enable)
ENABLE  PCA   Timer2  Serial  Timer1  Extrn1  Timer0  Extrn0
 (EA)   (EC)  (ET2)    (ES)    (ET1)   (EX1)   (ET0)   (EX0)
---------------------------------------------------------------
   1      0     1       0                        0       1
```

Execute the PROC.

```
hlt>iestat
                      IE REGISTER (Interrupt Enable)
ENABLE  PCA   Timer2  Serial  Timer1  Extrn1  Timer0  Extrn0
 (EA)   (EC)  (ET2)    (ES)    (ET1)   (EX1)   (ET0)   (EX0)
---------------------------------------------------------------
   1      0     1       0       1       0       1       1
```

2.5.4.2 Calling a PROC From a BRKREG

The following example illustrates using a BRKREG and PROC combination that ISTEPS through a program and displays the last instruction executed. Based on a user response, it either continues to step or halts emulation.

```
hlt>DEFINE PROC stepper = DO
.hlt>REPEAT
..hlt>ISTEP                                   /* Step by machine instructions */
..hlt>PRINT NEWEST                            /* Print last command executed */
..hlt>WRITE 'Continue?'
..hlt>DEFINE CHAR response = CI               /* CI is a built-in function that */
..hlt>WRITE response                          /* reads a character from console */
..hlt>IF (response == 'n') OR (response == 'N') THEN
...hlt>RETURN TRUE
...hlt>ELSE RETURN FALSE
...hlt>ENDIF
..hlt>ENDREPEAT
.hlt>END
hlt>
```

The following BRKREG calls stepper whenever the program is executing outside a range of instructions (range break).

```
    hlt> DEFINE BRKREG rangebreak = OUTSIDE #24 LENGTH 10 CALL stepper
    hlt>
```

Execute the BRKREG and PROC.

```
hlt> GO FROM 0 USING rangebreak
emu>
    0006H 75D000    MOV    PSW,#00H
Last instruction executed
FRAME     ADDR  CODE       INSTRUCTION
(01)      0003H 758155     MOV     SP,#55H     ; +085T
Continue?
Y
    008AH 110E        ACALL   (INIT)
Last instruction executed
FRAME     ADDR  CODE       INSTRUCTION
(FD)      0009H 02008A     LJMP    (:MAIN_DISPLAY#01)
Continue?
n
Probe stopped at 0009H (:MAIN_DISPLAY#13) because of internal break.
Break Register is [RANGEBREAK]
hlt>
```

NOTE

When a PROC is called from a BRKREG, it must return either TRUE or FALSE to the emulation processor. RETURN TRUE halts emulation; RETURN FALSE causes emulation to resume.

PROCs are not executed under control of the emulation processor. Therefore, HALT does not stop the execution of a PROC. Use CNTL-C (or CNTL-BREAK) to abort the execution of a PROC.

2.5.4.3 Using the WAIT Command in PROCs

The WAIT command needs to be used in PROCs that contain GO commands. WAIT suspends command processing until the emulation processor has finished emulation. The following PROC illustrates the use of the WAIT command.

```
hlt> DEFINE PROC go_and_wait
.hlt> REPEAT
..hlt> GO TIL #24
..hlt> WAIT
..hlt> UNTIL DPTR == 0C0DH
..hlt> ENDREPEAT
.hlt> END
hlt>
```

Refer to the PROC entry in Chapter 3 for more information on using PROCs as well as learning how to pass parameters to and from PROCs. The ICE-5100 tutorial makes extensive use of PROCs in displaying and debugging the sample MESSG program.

2.6 Controlling Trace Collection

This section describes how to enable and disable trace collection. By default, the ICE-5100 emulator always traces program execution. The ICE-5100 emulator can store 254 frames of trace data in its trace buffer. IF the buffer overflows, the oldest frames are replaced with new data as it is collected.

Use the TRACE option with the GO command to selectively trace your program's execution. For example, the following command causes TRACE to be turned off until program execution reaches line #24.

```
hlt> GO FROM 0 TRACE FROM #24
emu>
```

Trace only a range of addresses in your program.

```
hlt> GO FROM 0 TRACE #24 TO #33
emu>
```

Use the OUTSIDE option to gather TRACE data when the program is outside a specified address range.

```
hlt> GO FROM 0 TRACE OUTSIDE #24 LENGTH 6
emu>
```

2.6.1 Types of Tracepoints Allowed

The ICE-5100 emulator supports three types of tracepoints as described below:

- Specific trace — Traces a specific instruction address.
- Range trace — Traces instruction addresses within a specified range of addresses.
- Page trace — Traces instruction addresses within a page (256 bytes).

Only one type of tracepoint is allowed per GO command.

2.6.2 Using TRCREGs

A TRCREG contains a trace specification which is used in a GO command.

2.6.2.1 Creating TRCREGs

The following examples define TRCREGs.

```
hlt> DEFINE TRCREG trc1 = FROM #24 TIL #33
hlt> DEFINE TRCREG trc2 = 50 TO 90
hlt> DEFINE TRCREG trc3 = OUTSIDE PAGE 3
hlt>
```

2.6.2.2 Executing TRCREGs

Execute TRCREGs by specifying the TRCREG name after entering the GO TRACE command. For example:

```
hlt> GO FROM 0 TRACE trc1
emu>
```

2.6.2.3 Removing Tracepoints

As with breakpoints, the ICE-5100 emulator sets hardware tracepoints when you specify a TRCREG or tracepoint with the GO command. These tracepoints remain set until you execute a GO TRACE command by itself (with no new trace specifications) or specify a new TRCREG with the GO TRACE command. The RESET ICE command also removes tracepoints.

The following example enables full-time collection of trace data.

```
hlt> GO FROM 0 TRACE
emu>
```

2.7 Displaying Trace Data

Use the PRINT command to display selected frames of trace data. Options are available which enable you to display the oldest, newest, last, or next frame(s) of trace data.

If you are using the CLIPS assembly (refer to Appendix C), and you want to display the clips data, use the PRINT CLIPS option (by default, the display of clips is initially turned off).

Display the trace frame currently pointed to by the trace pointer.

```
hlt> PRINT
FRAME      ADDR    CODE        INSTRUCTION
(00)       0091H   80F9        SJMP    (#37)       ; $ - 05H
hlt>
```

Display the entire contents of the trace buffer (example shows only a partial display).

```
hlt> PRINT ALL
FRAME     ADDR    CODE      INSTRUCTION
(00)      0091H   80F9      SJMP     (#37)      ; $ - 05H
(01)      008CH   300002    JNB      INT_FLAG (#40) ; $ + 05H
(02)      0091H   80F9      SJMP     (#37)      ; $ - 05H
(03)      008CH   300002    JNB      INT_FLAG (#40) ; $ + 05H
(04)      0091H   80F9      SJMP     (#37)      ; $ - 05H
(05)      008CH   300002    JNB      INT_FLAG (#40) ; $ + 05H
(06)      0091H   80F9      SJMP     (#37)      ; $ - 05H
(07)      008CH   300002    JNB      INT_FLAG (#40) ; $ + 05H
(08)      0091H   80F9      SJMP     (#37)      ; $ - 05H
(09)      008CH   300002    JNB      INT_FLAG (#40) ; $ + 05H
(0A)      0091H   80F9      SJMP     (#37)      ; $ - 05H
(0B)      008CH   300002    JNB      INT_FLAG (#40) ; $ + 05H
hlt>
```

Display the clips data along with the trace data.

```
hlt> PRINT CLIPS
FRAME      ADDR    CODE        INSTRUCTION      CLIPS(76543210)
(0C)       0091H   80F9        SJMP    (#37)          10101111
hlt>
```

Display the oldest frame in the trace buffer (CLIPS display remains on).

```
hlt> PRINT OLDEST
FRAME      ADDR    CODE        INSTRUCTION      CLIPS(76543210)
(0C)       0091H   80F9        SJMP    (#37)          10101111
hlt>
```

Turn off the display of clips data with the PRINT NOCLIPS command option. Clips display remains off until you re-execute a PRINT CLIPS command.

```
hlt> PRINT NOCLIPS LAST
FRAME      ADDR    CODE        INSTRUCTION
(04)       0091H   80F9        SJMP    (#37)       ; $ - 05
hlt>
```

Display a specified trace frame (frame specified must be a number or expression evaluating to a number between 0 and 253T).

```
hlt> PRINT 10
FRAME        ADDR    CODE        INSTRUCTION
(0A)         0091H   80F9        SJMP    (#37)     ;$-05
hlt>
```

2.7.1 Clearing the Trace Buffer

Use the PRINT CLEAR command to clear the trace buffer.

```
hlt> PRINT CLEAR
hlt>
```

NOTE

Changing the current execution point causes the trace buffer to be cleared upon execution of the next GO command. If you so desire, use the EDIT TRACE command to save the trace buffer contents prior to changing the execution point (refer to the EDIT entry in Chapter 3).

2.8 Saving Debug Objects to Disk Files

There are two commands available which enable you to save and append debug objects to disk files for use in later debugging sessions. Use the PUT command to create a file and then save debug objects to it. Use the APPEND command to add debug objects to an existing file.

2.8.1 Using the PUT Command

Use the PUT command to create a disk file and save debug objects to it. You can globally save all currently defined debug objects with the DEBUG option, by type of debug object (i.e., LITERALLY, TRCREG, PROC, BRKREG), by *mtype*, or individually (e.g., brk1). When saving *mtype* debug objects, the value of the debug object is not saved.

If the specified file already exists, you are prompted as to whether or not you want to overwrite it. Any response other than y (or Y) aborts the PUT command.

Save all currently defined debug objects on an IBM host.

```
hlt> PUT C:\scrapdir\debug.lst DEBUG
hlt>
```

Save all currently defined BRKREGs on an Intel host.

```
hlt> PUT :F1:brks.lst BRKREG
hlt>
```

Save all currently defined BYTE *mtypes*.

```
hlt> PUT C:\scrapdir\lst.fil BYTE
hlt>
```

Save an individual debug object.

```
hlt> PUT C:\scrapdir\lst.fil stepper
hlt>
```

2.8.2 Using the APPEND Command

Use the APPEND command to append debug objects to an existing file. If the file exists, the specified debug objects are saved to it; otherwise APPEND creates the file and then saves the debug objects.

Command options are the same as those for the PUT command.

Append a debug object to an existing file.

```
hlt> APPEND C:\scrapdir\lst.fil brk2
hlt>
```

2.9 Removing Debug Objects from Memory

Use the REMOVE command to remove debug object definitions. At times you may have to remove unnecessary debug objects from memory (i.e., when you have used up all available host memory space).

Command options are the asme as those for the PUT and APPEND commands.

Remove all currently defined debug objects from memory.

```
hlt> REMOVE DEBUG
hlt>
```

2.10 Saving Mapped Program Memory to Disk Files

The SAVE command enables you to save segments of program executable code to a disk file.

Save a block of program memory to a file.

```
hlt> SAVE C:\scrapdir\sav.log 0 LENGTH 10K
hlt>
```

Restore a previously saved segment of program code memory with the LOAD command.

```
hlt> LOAD C:\scrapdir\sav.log
hlt>
```

2.11 Where to Find Additional Information on ICE™-5100 Commands

Chapters 1 and 2 have introduced the more important ICE-5100 emulator commands. There are many useful commands that have not been described in these chapters. Refer to Table 3-1 in Chapter 3 for a complete listing of ICE-5100 commands and a brief introduction to each command's purpose. Chapter 3 contains an entry on every command as well as entries listing ICE-5100 emulator keywords and other related topics.

First time users should go through the ICE-5100 emulator tutorial. It provides over 100 easy to use on-line screens describing the ICE-5100 emulator. It teaches you to debug a sample PL/M-51 program, offers a sample macro file for your use, and provides information on creating and using emulator commands.

Appendixes A through G present additional information on a wide range of topics that relate to using the ICE-5100 emulator. The information is presented in a condensed and easy to read manner.

User probe related information is provided in your user probe supplement; installation and host configuration in the *ICE™-5100 Emulator Installation Supplement*, order number 167095.

Use the pocket reference after becoming familiar with the commands and their purpose. Pocket references are provided for each user probe and contain ICE-5100 emulator commands and user probe commands. Also included is a list of probe-specific keywords and a list of special function registers and bits.

Command Encyclopedia 3

166267-001

3 COMMAND ENCYCLOPEDIA CONTENTS

Page

3 COMMAND ENCYCLOPEDIA

3.1 Introduction

This chapter contains the ICE-5100 emulator commands and related topics in alphabetical order. There are six categories under which emulator commands are grouped. To help you become familiar with the commands, Table 3-1 lists the categories and the commands within each group.

ICE-5100 emulator commands are printed in uppercase letters (e.g., ADDRESS), while only the first letter of each topic is capitalized (e.g., Addr-spec).

3.2 Syntax Notation

The following syntax notation is used throughout this chapter. Becoming familiar with the notation makes using the command options easier.

italics	indicate variable expressions. Substitute a value or symbol.
{*items*}	between braces indicate you must select one and only one item.
[*items*]	between brackets are optional items. You may select more than one item.
. . .	the preceding item may be repeated.
,. . .	the preceding item may be repeated with a comma between each repetition.
( )	are necessary to enclose an expression such as (#35 + 5).
device	stands for the number or letter of a disk drive.
dirname	stands for any user-created directory.
pathname:	is the fully-qualified reference to a file.
'	denotes your keyboard's apostrophe (') key. If your keyboard has two apostrophe keys, determine which one the ICE-5100 emulator accepts in command syntax.
<CNTL>	denotes the host keyboard's control key. For example, <CNTL><C> means enter C while pressing the control key.
<RETURN>	denotes the carriage return key.
`shading`	indicates user input.
punctuation	other than ellipses (. . .), braces ({ }), and brackets ([ ]) must be entered as shown.

Table 3-1 ICE™-5100 Emulator Commands Grouped by Category

Command	Description
Emulation and Trace Commands	
BRKREG	Register used to store break specifications.
CAUSE	Displays why and where emulation halted.
GO	Starts program execution; controls break and trace functions.
HALT	Halts program emulation.
ISTEP	Single-steps through a program by machine language instructions.
LSTEP	Single-steps through a program by line numbers.
PRINT	Displays the contents of the trace buffer; optionally displays clips data.
SYNCSTART	Enables and disables multi-ICE™ emulation.
TRCREG	Register used to store trace specifications.
WAIT	Suspends command processing during emulation.
$	Displays or changes the current execution point; references program counter.
Memory Access Commands	
ADDRESS	Displays or changes memory as 16-bit unsigned values.
ASM	Displays or changes program memory as ASM-51 assembly mnemonics.
BIT	Displays or changes bit memory as 1-bit values.
BOOLEAN	Displays or changes memory as Boolean values.
BYTE	Displays or changes memory as 8-bit unsigned values.
CHAR	Displays or changes memory as ASCII characters.
LOAD	Loads a program into mapped program memory.
MAP	Displays or sets the memory map.
SAVE	Saves portions of program memory to a file.
VERIFY	Specifies memory write verification.
WORD	Displays or changes memory as 16-bit unsigned values.
Debug Environment Commands	
APPEND	Appends debug object defintions to a file.
BASE	Displays or changes the number base.

Table 3-1 ICE™-5100 Emulator Commands Grouped by Category (continued)

Command	Description
Debug Environment Commands (cont.)	
DEFINE	Defines a debug object.
DIR	Displays program symbols and debug object names.
ERROR	Enables and disables the display of error information.
EVAL	Calculates and displays the result of an expression.
EXIT	Terminates a debug session and returns control to the host operating system.
INCLUDE	Retrieves command defintions from a file.
LIST/NOLIST	Opens or closes a list file.
LITERALLY	Stores and displays user-defined character strings.
MENU	Enables and disables the syntax menu display.
Paging	Controls screen display.
PROC	Stores a collective group of emulator commands.
PUT	Saves debug object defintions to a file.
REMOVE	Removes debug object definitions from memory.
WRITE	Formats and displays user-defined text to the screen.
String Functions	
CI	Reads a character from the host console.
CONCAT	Concatenates two strings into a single string and displays the result.
DCI	Reads a character from the host console; returns a zero if no character entered.
INSTR	Finds the starting position of a substring within a string.
NUMTOSTR	Converts an expression into ASCII code.
STRLEN	Returns the length of a character string.
STRTONUM	Converts a string to a numeric value.
SUBSTR	Extracts a portion of a character string.
Utility Commands	
EDIT	Invokes the debug editor.
CLEAREOL	Clears the screen from the cursor to end of line.
CLEAREOS	Clears the screen from the cursor to end of screen.
CURHOME	Moves cursor to top left-hand corner of the screen.

Table 3-1 ICE™-5100 Emulator Commands Grouped by Category (continued)

Command	Description
Utility Commands (cont.)	
CURX	Displays the column number or moves the cursor to column *x*.
CURY	Displays the row number or moves the cursor to row *y*.
DPTR	Displays or changes the DPTR register.
DYNASCOPE	Controls the operation of NAMESCOPE.
HELP	Provides on-line help.
INT	Displays interrupts that were active when emulation was halted.
NAMESCOPE	Displays or sets the current namescope for symbolic references.
RBANK	Displays or selects a bank of general purpose registers.
REGS	Displays selected registers and flags.
RESET	Resets specific emulator functions.
R0 - R7	Displays or modifies general-purpose registers.
SYMBOLIC	Enables or disables symbolic display.
TEMPCHECK	Controls user probe temperature checking.
TM0 - TM2	Displays or modifies the timer/counter registers.
VERSION	Displays the software version numbers
Compound Commands	
COUNT	Groups and executes emulator commands a specified number of times.
DO	Groups and executes commands.
IF	Groups and conditionally executes commands.
REPEAT	Groups and executes commands endlessly or until an exit condition is met.

3.3 ICE™-5100 Emulator Commands, Keywords, and Topics

The ICE-5100 commands, keywords, and related topics are listed alphabetically in this section. Each entry has the following format: The name of the command, keyword, or related topic, a brief description and syntax (where applicable), a discussion of its use, examples, and cross-references.

$

Displays or changes
the program counter

Syntax

$ [= *addr-spec*]

Where:

$	displays the contents of the program counter.
addr-spec	is a valid address or memory location. The address can be numeric or symbolic.

Discussion

The dollar sign ($) represents the program counter or fetch address of the next instruction. Use the dollar sign to display or change the current execution point.

NOTE

Changing the current execution point causes the next GO, ISTEP, or LSTEP command to begin with an empty trace buffer. If you want to save the contents of the trace buffer, use the EDIT TRACE command to save the trace data to a disk file.

Examples

1. Display the contents of the program counter.

```
hlt>$
05BAH
```

2. Modify the address in the program counter using symbolic and numeric references.

```
hlt>$ = coldstart + 4BH
hlt>$ = 0ABH
```

3. Save the contents of the program counter as a variable.

```
hlt>DEFINE ADDRESS start = $
```

Cross-Reference

GO
ISTEP
LSTEP
Pseudo-variable
REGS

ADDRESS

Displays or changes memory as a 16-bit value

Syntax

$$\text{ADDRESS [}\textit{mspace}\text{] }\textit{addr-spec}\left[= \left\{ \begin{array}{l} \textit{expression} \\ \textit{mtype}\text{ [}\textit{mspace}\text{] }\textit{addr-spec} \end{array} \right\} \left[\ , \ldots \right] \right]$$

Where:

ADDRESS	is the keyword to display or write to a memory location.
mspace	is IDATA, XDATA, RDATA, or CODE.
addr-spec	is a memory location or range of locations.
expression	must convert to a 16-bit unsigned value.
mtype	is ADDRESS, BIT, BYTE, CHAR, or WORD.

Discussion

Use ADDRESS to read or write the contents of memory. There are three operations that ADDRESS can perform: read from memory, write to memory, and copy from one memory location to another. ADDRESS is a memory object type (refer to the Mtype entry in this chapter), so it can also be used to define debug objects. ADDRESS has the same function as WORD. Use it in place of WORD to make debug procedures self-documenting.

Examples

1. Set a single value in XDATA, then display the value.

```
hlt> ADDRESS XDATA 0083H = 10FA
hlt> ADDRESS XDATA 0083H
XDATA 0083H 10FA
hlt>
```

2. Copy a single value from one memory location in IDATA to another.

```
hlt> ADDRESS 0021H = ADDRESS 0023H
hlt>
```

3. Define an ADDRESS debug object in IDATA with an initial value = 10FA.

```
hlt> DEFINE ADDRESS bench1 = 10FA
hlt>
```

Cross-Reference

DEFINE
Mspace
Mtype

Addr-spec

Is an address specification

Syntax

addr-spec is:

```
⎧  ⎧ address                  ⎫                                       ⎫
⎪  ⎨ address TO address       ⎬                                       ⎪
⎪  ⎩ address LENGTH number    ⎭                                       ⎪
⎪                                                                     ⎪
⎨  ⎡ [.]:module    ⎤                      ⎡ .label    ⎤               ⎬
⎪  ⎢               ⎥ [.procedure][, . . .] ⎢ #line     ⎥               ⎪
⎪  ⎣ [.]:procedure ⎦                      ⎣ .variable ⎦               ⎪
⎪                                                                     ⎪
⎩   expression                                                        ⎭
```

Where:

address	is a location in memory.
number	is a number or expression that evaluates to an integer between zero and +65,535.
module	is the name of a program or module.
procedure	is the name of a program procedure.
label	is the name of a program label.
line	is a program line number.
variable	is the name of a program variable.
expression	is an expression that evaluates to an address.
.	is the dot operator. The dot operator indicates the address of a variable is to be returned rather than the value.

Discussion

An *addr-spec* refers to an address, or a range of addresses. Addresses reference program or memory locations. Some ICE-5100 emulator commands require one address, some will accept a range of addresses. The syntax qualifications for commands using *addr-spec* note the specific meaning. A valid address may be either symbolic or numeric.

An address is a valid numeric location for the ICE-5100 emulator, or a program symbol proceded by a dot (.). When using numeric addresses, be aware of the BASE. An address can be specified in any base (hexadecimal, decimal, or binary), but is displayed in hexadecimal.

Several ICE-5100 emulator commands require an address value as shown in the following examples.

```
hlt> $ = addr-spec
hlt>

hlt> NAMESCOPE = addr-spec
hlt>

hlt> GO FROM addr-spec TIL addr-spec TRACE addr-spec TO addr-spec
hlt>

hlt> MAP addr-spec LENGTH number ICE
hlt>
```

Examples

1. Assign a new address to the program counter.

```
hlt> $ = 06H
hlt> $
0006
hlt>
```

2. Symbolic references evaluate to addresses and need no conversion. For example:

```
hlt> $ = :main_display                 /*Symbolic reference to the
hlt>                                      beginning of a module*/

hlt> $ = :main_display.rotate          /*Symbolic reference to the
hlt>                                   beginning of a procedure*/

hlt> $ = :main_display#24              /*Symbolic reference to a
hlt>                                      program line number*/
```

Cross-Reference

Expression
Mtype
Symbolic references

APPEND

Adds debug object definitions to a file

Syntax

```
                          ┌ DEBUG                               ┐
                          │        ┌ BRKREG    ┐                 │
                          │        │ TRCREG    │                 │
APPEND pathname          <        <  PROC       >  [, . . .]     >
                          │        │ LITERALLY │                 │
                          │        │ mtype     │                 │
                          └        └ name      ┘                 ┘
```

Where:

APPEND	opens the file specified in *pathname* to store debug objects.
pathname	is the fully-qualified reference to the file into which you want to save debug objects.
DEBUG	appends all currently defined debug objects to the file named in *pathname*.
BRKREG . . . LITERALLY	limits objects appended to a file to the type specified.
mtype	is ADDRESS, BIT, BOOLEAN, BYTE, CHAR, or WORD.
name	is the user-defined name of a single debug object to be saved.

Discussion

Use the APPEND command to add current debug object definitions to an existing disk file. If the named file does not exist, APPEND creates it. If a debug object already exists in the APPEND file, both versions are saved. The values of mtype debug objects are not saved.

For use in future debugging sessions, use the INCLUDE command to retrieve the contents of the file. Only the most recent definition is restored by the INCLUDE command.

Examples

1. Define debug objects, save them in a new file, use them from the file.

```
hlt> DEFINE TRCREG trace1 = #12
hlt> DEFINE BRKREG brk1 = #23
hlt> APPEND dvars.01 trace1, brk1
hlt> INCLUDE dvars.01
DEFINE TRCREG trace1 #12
DEFINE BRKREG brk1 = #23
hlt>
```

2. Re-DEFINE a debug object, APPEND the new definition to the same file as the old, INCLUDE the file.

```
hlt> DEFINE BRKREG brk1 = #17
hlt> APPEND dvars.01 brk1
hlt> INCLUDE dvars.01
DEFINE TRCREG trace1 #12
DEFINE BRKREG brk1 = #23
DEFINE BRKREG brk1 = #17
hlt> BRKREG brk1
#17
hlt>
```

Cross-Reference

BRKREG
INCLUDE
LITERALLY
Mtype
PROC
PUT
REMOVE
TRCREG

ASM

Displays or modifies program memory
as ASM-51 assembly mnemonics

Syntax

```
                 ┌     ┌                                          ┐ ┐
ASM addr-spec    │  =  │ 'ASM-51 assembler-mnemonic' [, . . .]    │ │
                 │     │ <RETURN>                                 │ │
                 └     └                                          ┘ ┘
```

Where:

ASM	is the keyword. After a portion of memory has been disassembled, ASM can be entered without an *addr-spec* to disassemble the next instruction or range of instructions.
addr-spec	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *addr-spec* TO *addr-spec* or *addr-spec* LENGTH *number-of-instructions*.
ASM-51 assembly-mnemonic	is any ASM-51 assembly instruction except for CALL and JUMP.

Discussion

Use the ASM command to display or modify program memory. If the assignment portion of the command is not present, memory is displayed as ASM-51 assembly mnemonics. When the assignment portion is present, the ASM-51 assembly mnemonic is translated into the appropriate object code and placed into program memory at the specified *addr-spec*.

When using a range of addresses, be aware of instruction boundaries. Fill in with NOPs if necessary. Symbolic addresses must be fully qualified (:module.procedure.variable) when used inside an instruction. The example section explains more about using ASM.

NOTE

Disassembly with SYMBOLIC = TRUE is faster if the amount of program symbolic information is less than or equal to the current size of VSTBUFFER. (Refer to the ICE*nnn* entry in this chapter for details.)

Examples

1. **Disassemble code:** Disassemble program code using a symbolic reference to a memory location. Program disassembly displays a heading followed by the disassembled code.

```
hlt> ASM .:main_display.char_display
ADDR        CODE        INSTRUCTION
(:MAIN_DISPLAY.CHAR_DISPLAY)
005DH       22          RET
hlt>
```

Disassemble two instructions using the LENGTH option.

```
hlt> ASM .:main_display.rotate LENGTH 2
      ADDR        CODE        INSTRUCTION
(:MAIN_DISPLAY.ROTATE)
      005EH       750801      MOV         I,#01H      : +001T
(:MAIN_DISPLAY#27)
      0061H       852109      MOV         TEMP,DISP_BUFFER
hlt>
```

Disassemble a range of program memory using the TO option.

```
hlt> ASM #27 TO #29
      ADDR        CODE        INSTRUCTION
(:MAIN_DISPLAY #27)
      0061H       852109      MOV         TEMP, DISP_BUFFER
(:MAIN_DISPLAY #28)
      0064H       E508        MOV         A,I
      0066H       C3          CLR         C
      0067H       9571        SUBB        A,BUFF_SIZE
      0069H       5013        JNC         (#32)         :$ + 15H
(:MAIN_DISPLAY#29)
      006B        E508        MOV         A,I
hlt>
```

2. **Assemble code:** The following example assembles an ASM-51 mnemonic into program memory starting at location 0100H.

```
hlt> ASM 0100H = 'setb c'
hlt>
```

Assemble several instructions in program memory starting at 0100H.

```
hlt> ASM 0100H = 'mov a,i', 'clr c', 'subb a,temp'
hlt>
```

ASM (continued)

3. Assume a more complex instruction is to be patched. Use the ASM *addr-spec* = <return> form of syntax. When the ICE-5100 emulator responds with an *addr-spec*>, enter one mnemonic at a time.

```
hlt> ASM 0100H = <RETURN>
>>>>> Assembler Line Mode: enter blank line to terminate. <<<<<
0100H> mov a,i
0102H> clr c
0103H> subb a,temp
0105H> nop
0106> <RETURN>
hlt>
```

NOTE

If a patch requires more memory than is available within the program, replace the erroneous instruction with a jump to unused memory. The last instruction in the patch must be a return to the next instruction in the program.

Cross-Reference

Addr-spec
ICE*nnn*
SAVE
SYMBOLIC

BASE

Displays or changes the number base

Syntax

```
            ┌   ┌ expression ┐ ┐
            │   │ BINARY     │ │
BASE        │ = │ DECIMAL    │ │
            │   │ HEX        │ │
            └   └            ┘ ┘
```

Where:

BASE	displays the current number base; initial base is decimal.
expression	evaluates to decimal values 2, 10, or 16.
BINARY	changes the BASE to binary (base 2, suffix Y).
DECIMAL	changes the BASE to decimal (base 10, suffix T).
HEX	changes the BASE to hexadecimal (base 16, suffix H).

Discussion

The BASE pseudo-variable controls the number base. Use BASE as follows:

- To display the default base (e.g., BASE).
- To change to a new base (e.g., BASE = 2T or BASE = HEX).
- As a variable in expressions (e.g., *variable* = BASE). The memory type of the BASE pseudo-variable is BYTE.

When changing bases, *expression* must evaluate to the decimal values 2, 10, or 16. Otherwise, the number base does not change and an error results. Unless specified, all expressions are evaluated in the current base. Override the current base by using an explicit suffix.

NOTE

BASE is always global. When the number base is changed, the change happens immediately, even if the BASE command is within a debug procedure definition, a block command, or in a command line with multiple commands.

BASE (continued)

Examples

1. Display the current number base.

```
hlt> BASE
DECIMAL
hlt>
```

2. Change the number base to hexadecimal.

```
hlt> BASE = 10H
```

or

```
hlt> BASE=HEX
hlt>
```

3. Use BASE in an expression.

```
hlt> var1 = BASE * 2
hlt>
```

Cross-Reference

Expression
Pseudo-variable

BIT

Displays or changes bit memory
as one-bit binary values

Syntax

$$\text{BIT } \textit{addr-spec} \left[= \left\{ \begin{matrix} \textit{expression} \\ \textit{mtype [mspace] addr-spec} \end{matrix} \right\} \left[, \ldots \right] \right]$$

Where:

BIT	is the keyword to display or write to a memory location.
addr-spec	is a memory location or range of locations between addresses 00H - FFH.
expression	must convert to a one-bit binary value.
mtype	is ADDRESS, BIT, BOOLEAN, BYTE, or WORD.

Discussion

Use BIT to read or write the contents of bit memory. There are three operations that BIT can perform: read from memory, write to memory, and copy from one memory location to another. BIT is a memory object type (refer to the Mtype entry in this chapter), so it can also be used to define debug objects. BIT address 00H maps to the least significant bit at IDATA address 20H.

Examples

1. Display a range of bit memory space.

```
hlt> BIT 20H LENGTH 5
BIT   0020H    0 0 1 0 1
hlt>
```

2. Set a bit memory location.

```
hlt> BIT 20H = TRUE
```

or

```
hlt> BIT 20H = 1
hlt>
```

Cross-Reference

Mspace
Mtype

BOOLEAN

Displays or changes memory
as Boolean TRUE or FALSE values

Syntax

$$\text{BOOLEAN [}\textit{mspace}\text{] }\textit{addr-spec}\left[\; = \left\{\begin{array}{l}\textit{expression}\\ \textit{mtype}\text{ [}\textit{mspace}\text{] }\textit{addr-spec}\end{array}\right\}\left[\;,\ldots\;\right]\right]$$

Where:

BOOLEAN	is the keyword to display or write to a memory location.
mspace	is IDATA, XDATA, RDATA, or CODE.
addr-spec	is a memory location or range of locations.
expression	must convert to an 8-bit unsigned value.
mtype	is ADDRESS, BIT, BOOLEAN, BYTE, or WORD.

Discussion

Use BOOLEAN to read or write the contents of memory. There are three operations that BOOLEAN can perform: read from memory, write to memory, and copy from one memory location to another. BOOLEAN is a memory object type (refer to the Mtype entry in this chapter), so it can also be used to define debug objects. Only the least significant bit (LSB) of a BOOLEAN is tested. If the LSB is 1, the BOOLEAN value is true; if the LSB is 0, the BOOLEAN value is FALSE.

Examples

1. Set a single value in XDATA, then display the value.

```
hlt> BOOLEAN XDATA 0083H = 10FA
hlt> BOOLEAN XDATA 0083H
XDATA 0083H FALSE
hlt>
```

2. Copy a single value from one memory location in IDATA to another.

```
hlt> BOOLEAN 0051H = BOOLEAN 0083H
hlt>
```

3. Define a BOOLEAN debug object in IDATA with an initial value = TRUE.

```
hlt> DEFINE BOOLEAN bench1 = TRUE
hlt>
```

Cross-Reference

DEFINE
Mspace
Mtype

BRKREG

Defines or displays a register
containing break specifications

Syntax (two forms)

1. Define a BRKREG:

```
                              [DO]
DEFINE BRKREG name  =   ( [OUTSIDE] [PAGE addr-spec) [, . . .] [CALL proc]
                              [END]
```

2. Display a BRKREG:

```
BRKREG name
```

Where:

DEFINE	defines or redefines a debug object.
BRKREG	is the keyword for creating or displaying a break register.
name	is the name of a break register.
DO . . . END	encloses more than one command.
OUTSIDE	causes the ICE-5100 emulator to break emulation on all addresses other than those specified in *addr-spec* (a logical NOT function).
PAGE	specifies a range of addresses which is exactly 256 bytes in length. A page is specified with a single 8-bit page address. Valid page addresses range from 00H to 0FFH with PAGE 00H beginning at location 0000H.
addr-spec	is a memory location or range of locations.
CALL *proc*	calls the named debug procedure (PROC) when the break specification is met.

Discussion

Use a BRKREG to create and save break specifications.

The CALL option of the break registers allows you to call a debug procedure (PROC) that is executed when the address in the register is reached. The PROC must return TRUE for the emulator to remain in halt mode or FALSE indicating that the emulator should go back into emulation. The PROC is executed when any of the addresses in the break register with the CALL option are executed. NOTE: the ICE-5100 emulator is not in emulation when the PROC is being executed. (Refer to the PROC entry in this chapter for more information.)

Manipulating BRKREGs

Manipulate a BRKREG by referring to its name. You can manipulate BRKREGs in the following ways:

- Create BRKREGs with the DEFINE command.
- Delete BRKREGs from memory with the REMOVE command.
- List BRKREG names with the DIR command.
- Save BRKREGs with the PUT/APPEND commands.
- Restore BRKREGs from files with the INCLUDE command.
- Display BRKREGs with the BRKREG command.
- Execute BRKREGs with the GO USING command.
- Modify BRKREGs with the editor.

Examples

1. Define a BRKREG called brk1 that breaks when the first instruction of line 23 is executed.

```
hlt> DEFINE BRKREG brk1 = #23
hlt>
```

2. Use all the available breakpoints in one BRKREG. No other break points can be specified in the GO command. (Refer to the GO command for a list of valid breakpoint specifications.)

```
hlt> DEFINE BRKREG brk2 = :main.proc1, 100 TO 134
hlt>
```

3. Display a BRKREG definition.

```
hlt> BRKREG brk1
#23
hlt>
```

4. Define a BRKREG that breaks when a program is outside page 25 (addresses 2500H to 25FFH).

```
hlt> DEFINE BRKREG outpage = OUTSIDE PAGE 25
hlt> GO FROM 0 USING outpage
emu>
Probe stopped at 0452H because of internal break.
 Break Register is [OUTPAGE]
hlt>
```

5. Define a BRKREG that halts emulation when the program attempts to execute any instruction within a range of instructions.

```
hlt> DEFINE BRKREG rangebreak = 34 LENGTH 20
hlt> GO FROM 0 USING rangebreak
emu>
Probe stopped at 0043H because of internal break.
 Break Register is [RANGEBREAK]
hlt>
```

6. Define a BRKREG named this_round that calls the procedure query. The procedure displays the value of the registers and flags and asks if you want to stop program execution. Entering a Y returns TRUE to the calling BRKREG and stops execution.

```
hlt> DEFINE PROC query = DO
.hlt> REGS
.hlt> WRITE USING('"Do you want to break?",>')
.hlt> DEFINE CHAR ccc = CI
.hlt> WRITE ccc
.hlt> IF (ccc == 'Y') OR (ccc == 'y') THEN
..hlt> RETURN TRUE
..hlt> ELSE RETURN FALSE
..hlt> ENDIF
.hlt> END
hlt>
hlt> DEFINE BRKREG this_round = :main_display#33 CALL query
hlt>
hlt> GO FROM 0 USING this_round
emu>
$ = 0036H          TM0 = 0000H          R0 =02H          R4 = 0CH
ACC = 0AH          TM1 = 0000H          R1 = 02H          R5 = 01H
SP = 07H           TM2 = 0050H          R2 = 06H          R6 = 03H
DPTR = 10C6H       RBANK = 01H          R3 = B0H          R7 = 00H
FLAGS: CY = 1  AC = 0  F0 = 1  RS1 = 0  RS0 = 1  OV = 0  P = 0
Do you want to break?Y
Probe stopped at 005DH(:MAIN_DISPLAY#22)because of internal break.
 Break Register is[THIS_ROUND]
hlt>
```

Cross-Reference

APPEND
DEFINE
DIR
EDIT
GO
INCLUDE
Name
PROC
PUT
REMOVE

BYTE

Displays or changes memory
as an eight-bit value

Syntax

$$\text{BYTE [}\mathit{mspace}\text{] }\mathit{addr\text{-}spec} \left[= \left\{ \begin{array}{l} \mathit{expression} \\ \mathit{mtype}\ [\mathit{mspace}]\ \mathit{addr\text{-}spec} \end{array} \right\} \left[, \ldots \right] \right]$$

Where:

BYTE	is the keyword to display or write to a memory location.
mspace	is IDATA, XDATA, RDATA, or CODE.
addr-spec	is a memory location or range of locations.
expression	must convert to an 8-bit unsigned value.
mtype	is ADDRESS, BIT, BYTE, CHAR, or WORD.

Discussion

Use BYTE to read or write the contents of memory. There are three operations that BYTE can perform: read from memory, write to memory, and copy from one memory location to another. BYTE is a memory object type (refer to the Mtype entry in this chapter), so it can also be used to define debug objects. When BYTE is used to display memory, the value is shown in the current number BASE and as the corresponding ASCII characters enclosed in single quotes ('). A non-printing character is displayed as a period ('.').

Examples

1. Set a single value in XDATA, then display the value.

```
hlt> BYTE XDATA 0083H = 4A
hlt> BYTE XDATA 0083H
XDATA 0083H 4A                          'J'
hlt>
```

2. Copy a single value from one memory location in IDATA to another.

```
hlt> BYTE 0051H = BYTE 0083H
hlt>
```

3. Define a BYTE debug object in IDATA with an initial value = 4A.

```
hlt> DEFINE BYTE short1 = 4A
hlt>
```

Cross-Reference

DEFINE
Mspace
Mtype

CAUSE

Displays why and where emulation stopped

SYNTAX

CAUSE

Discussion

Use the CAUSE command to redisplay the cause of the last break in emulation. The message displayed is the same message that was displayed at the time emulation stopped. CAUSE is useful when the break message has scrolled off the screen due to subsequent user commands and screen displays.

If a break occurs while a PROC is executing, no break message is displayed. Use CAUSE to display break messages when the PROC has finished execution.

CAUSE displays one of four messages:

- Probe stopped at *addr-spec* because of internal break.
- Probe stopped at *addr-spec* because of external break.
- Probe stopped at *addr-spec* because of user halt.
- Probe stopped at *addr-spec* because of unknown cause.

The messages correspond to a break caused by a GO command, a GO command with SYNCSTART enabled, a HALT command, or an unknown cause, respectively.

Example

1. Display the reason why emulation halted.

```
hlt> CAUSE
Probe stopped at 0065H because of user halt.
hlt>
```

CHAR

Displays or changes memory
as an eight-bit ASCII character

Syntax

```
                            ┌                                                    ┐
                            │    ┌ expression                  ┐ ┌       ┐       │
CHAR [mspace] addr-spec     │ = <                               > │ , . . . │       │
                            │    └ mtype [mspace] addr-spec    ┘ └       ┘       │
                            └                                                    ┘
```

Where:

CHAR	is the keyword to display or write to a memory location.
mspace	is IDATA, XDATA, RDATA, or CODE.
addr-spec	is a memory location or range of locations.
expression	must convert to an 8-bit unsigned value.
mtype	is ADDRESS, BYTE, CHAR, or WORD.

Discussion

Use CHAR to read or write the contents of memory. There are three operations that CHAR can perform: read from memory, write to memory, and copy from one memory location to another. CHAR is a memory object type (refer to the Mtype entry in this chapter), so it can also be used to define debug objects. When used for memory display, CHAR prints each byte of the specified memory location as an ASCII character, enclosing the entire string in single quotes ('). A non-printing character is displayed as a period ('.').

Examples

1. Set a single value in XDATA, then display the value.

```
hlt> CHAR XDATA 0083H = 'J'
hlt> CHAR XDATA 0083H
XDATA 0083H 'J'
hlt>
```

2. Copy a single value from one memory location in IDATA to another.

```
hlt> CHAR 0051H = CHAR 0083H
hlt>
```

3. DEFINE a CHAR debug object in IDATA with an initial value = 10FA.

```
hlt> DEFINE CHAR bench1 = 'J'
hlt>
```

CHAR (continued)

Cross-Reference

DEFINE
Mspace
Mtype

Reads one character
from the host console

Syntax

CI

Discussion

The CI (console input) function is useful inside a PROC requiring interaction with the console. CI reads one character from the host console. The emulator pauses until the character is entered. No prompt is displayed while the emulator is waiting for the character, and the entered character is not echoed to the screen. No carriage return is required after the character has been entered.

Example

1. The following example illustrates the use of the CI command within a debug procedure.

```
hlt> DEFINE PROC test = DO
.hlt> REPEAT
..hlt> LSTEP
..hlt> WRITE'Continue?'
..hlt> DEFINE CHAR reply = CI
..hlt> WRITE reply
..hlt> UNTIL (reply == 'n') OR (reply == 'N')
..hlt> ENDREPEAT
.hlt> WRITE'Done!'
.hlt> END
hlt>
```

2. The following example assumes the program has stopped just before line number 1 and now the programmer will use the test procedure to LSTEP and then query for continuation.

```
hlt> test
:MAIN_DISPLAY#3
Continue?
y
:MAIN_DISPLAY#6
Continue?
n
Done!
hlt>
```

Cross-Reference

DCI

CLEAREOL

Clears the screen from the cursor to the end of the line

Syntax

CLEAREOL

Discussion

The CLEAREOL command clears the screen display from the cursor's location *after* the command is entered to the end of the line. CLEAREOL is useful in debug procedures to manipulate screen output.

Examples

1. Clear the first line of the screen display. The <RETURN> key moves the cursor to the next line; then the cursor is moved to the top left-hand corner of the screen by CURHOME, after which CLEAREOL is executed.

```
hlt> CURHOME; CLEAREOL <RETURN>
hlt>
```

2. Clear the second line of the screen display. CURHOME moves the cursor to the top left-hand corner of the screen; then the <RETURN> key moves the cursor to the start of the second line before CLEAREOL is executed.

```
hlt> CURHOME <RETURN>
hlt> CLEAREOL <RETURN>
hlt>
```

Cross-Reference

CLEAREOS
CURHOME
CURX
CURY

CLEAREOS

Clears the screen from the cursor to the end of the screen

Syntax

CLEAREOS

Discussion

The CLEAREOS command clears the screen display from the cursor's location *after* the command is entered to the end of the screen. CLEAREOS is useful in debug procedures to manipulate screen output.

Examples

1. Clear the screen from the second line. CURHOME moves the cursor to the top left-hand corner of the screen; then the <RETURN> key moves the cursor to the second line before CLEAREOS is executed.

```
hlt> CURHOME <RETURN>
hlt> CLEAREOS <RETURN>
hlt>
```

2. Clear the screen from the first line. CURHOME moves the cursor to the top left-hand corner of the screen; then the <RETURN> key moves the cursor to the next line, after which CLEAREOS is executed.

```
hlt> CURHOME; CLEAREOS <RETURN>
hlt>
```

Cross-Reference

CLEAREOL
CURHOME
CURX
CURY

Command line editor

Enables you to edit and reuse commands

Table 3-2 summarizes the line edit functions. Refer to the *Debug Editor User's Guide*, order number 167098, for information on the debug editor.

Table 3-2 Command Line Editor Special Function Keys

Key Name	Function
<RUBOUT>	Deletes the character to the left of the cursor.
<CNTL><A>	Deletes the line to the right of the cursor.
<CNTL><C>	Cancels the command in progress. Use <CNTL><BREAK> on IBM hosts.
<CNTL><E>	Re-executes the last command.
<CNTL><F>	Deletes the character at the cursor and adjusts spacing.
<CNTL><X>	Deletes the line to the left of the cursor.
<CNTL><Z>	Deletes the current line.
<ESC>	Invokes the text editor and places the present command in the edit buffer for editing. If you press <ESC> at the prompt, the previous command is retrieved from the history buffer.
<←>	Moves the cursor left one character.
<→>	Moves the cursor right one character.
<↑>	Retrieves the previous line from the history buffer.
<↓>	Moves to the next line in the history buffer.
<HOME>	Magnifies the effect of the preceding arrow key. Causes jumps to the beginning or end of the current line when used with the <→> or <←> arrow.

Cross-Reference

Command recall
Edit

Command recall

Retrieves executed commands from the history buffer

Syntax

$$\left\{ \begin{array}{c} \uparrow \\ \downarrow \end{array} \right\}$$

Where:

↑ and ↓	are the up-arrow and down-arrow cursor keys.

Discussion

The ↑ and ↓ cursor keys enable you to retrieve previously executed commands from the 400-byte history buffer.

Press the ↑ key to retrieve the latest command. Repeatedly pressing the ↑ key moves you backward through the history buffer. To move forward again, press the ↓ key.

You can modify an old command with the line editor and then press the <RETURN> key from anyplace on the command line to execute the command. The executed command becomes the latest command in the history buffer.

Cross-Reference

Command line editor
Control keys
EDIT

Comments

Enables you to comment debug object definitions, list files, etc.

Syntax

/ user comments */*

Discussion

Any text enclosed in comment markers (/* */) will not be recognized or executed by the ICE-5100 emulator software. Use comment markers to enclose notes in debugging log files, or in PROCs.

Example

```
hlt> DEFINE PROC ask = DO
.hlt> WRITE 'Do you want to resume emulation?'
/* Ask user a question */
.hlt> DEFINE CHAR reply = CI            /* Get response from console */
.hlt> IF (reply == 'Y') OR (reply == 'y') THEN
..hlt> RETURN FALSE                             /* Resume emulation */
..hlt> ELSE
.hlt> RETURN TRUE                             /* Else stop emulation */
.hlt> END IF
.hlt> END
hlt>
```

CONCAT

Creates and displays concatenated strings

Syntax

CONCAT (*string-ref* [, . . .])

Where:

string-ref	is characters enclosed in apostrophes, a string expression using CONCAT, NUMTOSTR, or SUBSTR functions, or a reference to a CHAR type debug object.

Discussion

The CONCAT function builds new strings by concatenating all or parts of old strings. CONCAT is useful in a debug procedure (PROC). A debug procedure saves the construction and prints it when the procedure is executed.

Use the CONCAT command to display or save a concatenated message. When the CONCAT command is issued at the prompt, it displays the new message immediately but does not save the new construction.

Examples

1. Concatenate two strings, the predefined character string msg1 and the literal string 'PROC1':

```
hlt>DEFINE CHAR msg1 = 'Now executing '
hlt>CONCAT (msg1, 'Proc1')
Now executing Proc1
hlt>
```

2. Concatenate two strings inside a debug object definition.

```
hlt>DEFINE CHAR msg2 = CONCAT (msg1, 'Test Procedure')
hlt>msg2
Now executing Test Procedure
hlt>
```

Cross-Reference

CHAR
INSTR
NUMTOSTR
Strings
STRLEN
STRTONUM
SUBSTR

Control keys

List of control keys recognized by the ICE-5100 emulator

Table 3-3 lists the ICE-5100 emulator control keys and their functions.

Table 3-3 The ICE™-5100 Emulator Control Keys

Key Name	Function
<CNTL>A	Deletes the line to the right of the cursor.
<CNTL>C	Cancels the command in progress. Use <CNTL><BREAK> on IBM hosts.
<CNTL>D	Enters the ISIS Debug-86 monitor on the Series III. Type G <RETURN> to return to the debug environment.
<CNTL>E	Re-executes the last command
<CNTL>F	Deletes the character at the cursor and adjusts spacing.
<CNTL><NUMLOCK>	Stops the screen display on an IBM PC AT or PC XT. Enter any key to resume the display.
<CNTL>Q	Resumes the screen display halted by <CNTL>S on a Series III.
<CNTL>S	Stops the screen display on a Series III.
<CNTL>V	Toggles the menu display.
<CNTL>X	Deletes the line to the left of the cursor.
<CNTL>Z	Deletes the current line.

Cross-Reference

Paging

COUNT

Groups and executes commands
a specified number of times

Syntax

```
COUNT expression
    [ ICE-5100 emulator command(s) ]
    [ WHILE boolean-condition      ]
    [ UNTIL boolean-condition      ]

END [COUNT]
```

Where:

COUNT	is a repetition keyword.
expression	specifies the maximum number of times the COUNT command loop executes. The *expression* must evaluate to a positive whole number, less than or equal to 65535 in the current base.
ICE-5100 emulator command(s)	are all ICE-5100 emulator commands except EDIT, HELP, and INCLUDE.
WHILE	executes the COUNT loop while *boolean-condition* is TRUE. Execution halts when the WHILE condition is FALSE (unless the terminal count is reached first).
UNTIL	halts COUNT loop execution when *boolean-condition* is TRUE (unless the terminal count is reached first).
boolean-condition	evaluates to TRUE (LSB = 1), or FALSE (LSB = 0).
END[COUNT]	terminates the COUNT block. The optional COUNT keyword labels the block type.

Discussion

Unless it is within a debug procedure definition, a COUNT block is executed immediately after you enter the END statement.

COUNT blocks not containing WHILE or UNTIL clauses are executed at least once. COUNT blocks containing WHILE or UNTIL exit when the test condition is satisfied or the count value is reached, whichever occurs first.

Example

1. The following example shows an UNTIL clause that exits the block before the terminal count is reached. The example assumes that the number base is hexadecimal.

```
hlt>DEFINE BYTE bb
hlt>bb = 0
hlt>COUNT 10
.hlt>UNTIL bb == 5
.hlt>bb
.hlt>bb = bb + 1
hlt>END
00
01
02
03
04
hlt>
```

Cross-Reference

Expression
PROC
REPEAT

CURHOME

Moves the cursor to home position

Syntax

CURHOME

Discussion

The CURHOME command moves the cursor to the top left corner of the display screen. CURHOME is useful in debug procedures to manipulate screen output.

Example

1. Move the cursor to the top left-hand corner of the screen.

```
hlt> CURHOME
```

Cross-Reference

CLEAREOL
CLEAREOS
CURX
CURY
PROC
WRITE

CURX

Displays or modifies the current column position

Syntax

CURX [= *expression*]

Where:

CURX	displays the column number of the current cursor location in the current number base.
expression	ranges from 0 to the maximum number of columns on your CRT.

Discussion

The CURX pseudo-variable is typically used inside a debug procedure (PROC) with the CURY pseudo-variable to position the cursor on the display screen.

Any information written to the screen after the cursor is moved is written from the new cursor location. Any character previously displayed at that location is deleted from the screen as the new characters are written over the old.

If you access a column beyond the right border, the ICE-5100 emulator displays an exclamation point (!) in the last column and does not echo the display to the screen.

Example

1. This example shows cursor movement (from column 0 to column 40) after entering the CURX command.

```
hlt>CURX = 40T
                                        hlt>
```

Cross-Reference

CURHOME
CURY
Expression
Pseudo-variable

CURY

Displays or modifies the current row position

Syntax

CURY [= *expression*]

Where:

CURY	displays the row number of the current cursor location in the current number base.
expression	can be any decimal number and is evaluated to *expression* MOD 23.

Discussion

The CURY pseudo-variable is typically used in a debug procedure (PROC) with the CURX pseudo-variable to position the cursor on the display screen.

Any information written to the screen after the cursor is moved is written from the new cursor location. Any characters previously displayed at that location are deleted from the screen as the new characters are written over the old.

Example

1. This example shows cursor movement (from the first row to the fifth) in the vertical direction after entering the CURY command.

```
hlt> CURY = 5T




hlt>
```

Cross-Reference

CURHOME
CURX
Expression
Pseudo-variable

DCI

Reads one character
from the host console

Syntax

DCI

Discussion

The DCI (direct console input) function lets the ICE-5100 emulator read one character from the host console. With the DCI command, the emulator returns a null value (0) if there is no character in the keyboard buffer. A carriage return is not required after the character has been entered and the character is not echoed to the screen.

This function enables you wait for a character to be entered from the keyboard for a specified length of time rather than indefinitely, as with the CI function.

Example

1. The following example illustrates the use of the DCI command within a debug procedure.

```
hlt> DEFINE PROC query = DO
.hlt> DEFINE BYTE cnt = 0
.hlt> DEFINE CHAR chr = ' '
hlt> WRITE USING (' "Continue?" ,>')
.hlt> REPEAT
..hlt> chr = DCI                          /* Read a character */
..hlt> cnt = cnt + 1                  /* Increment time value */
..hlt> UNTIL cnt ==100H OR chr <> 0
..hlt> ENDREPEAT
.hlt> IF chr == 'y' OR chr == 'Y' THEN
..hlt> WRITE chr
..hlt> RETURN TRUE
..hlt> ELSE
..hlt> RETURN FALSE
..hlt> END IF
.hlt> END
hlt>
```

Execute PROC:

```
hlt> query
Continue? Y
TRUE
hlt>
```

Cross-Reference

CI

DEFINE

Defines a debug object

Syntax (four forms)

1. Define a LITERALLY:

```
DEFINE LITERALLY name = 'character-string'
```

2. Define a debug procedure:

```
DEFINE PROC name = DO
                       ICE-5100 emulator command(s)
                   END
```

3. Define a debug register:

```
        ┌                                                                          ┐
        │                  [DO]                                                    │
        │ BRKREG name =    ( [OUTSIDE] [PAGE] addr-spec) [, . . .] [CALL proc]     │
        │                  [END]                                                   │
DEFINE  │                  [DO]                                                    │
        │                    ┌ [OUTSIDE] [PAGE] addr-spec            ┐             │
        │ TRCREG name =      └ [FROM addr-spec] [TIL addr-spec]      ┘             │
        │                  [END]                                                   │
        └                                                                          ┘
```

4. Define a debug object:

```
DEFINE [GLOBAL] mtype name [ = expression]
```

Where:

DEFINE	defines or redefines a debug object.
LITERALLY	specifies that a LITERALLY is to be defined.
PROC	specifies that a debug procedure is to be defined.
BRKREG	specifies that a break register is to be defined.
TRCREG	specifies that a trace register is to be defined.
mtype	is ADDRESS, BIT, BOOLEAN, BYTE, CHAR, or WORD.
GLOBAL	signals that a *mtype* debug object is to be global rather than local to a block.
expression	evaulates to a valid value for the specified *mtype*.

Discussion

The define command lets you define debug objects that customize a debugging session. Execute them using their names. Save debug objects with the PUT and APPEND commands. Recall them from a file with the INCLUDE command. Remove them from memory with the REMOVE command.

Cross-Reference

Addr-spec
APPEND
BRKREG
Expression
LITERALLY
Mtype
Name
PROC
PUT
REMOVE
TRCREG

DIR

Displays program symbols and debug object names

Syntax

```
       ┌ ┌          ┌ ┌ mtype      ┐ ┐          ┐
       │ │          │ │ BRKREG     │ │          │
       │ │ DEBUG    │ { LITERALLY  } │          │
       │ │          │ │ PROC       │ │          │
       │ │          └ └ TRCREG     ┘ ┘          │
       │ │                                      │
       │ │            ┌ ┌ mtype     ┐ ┐         │
DIR    │ { PUBLICS    │ { LABEL     } │         }
       │ │            └ └ PROCEDURE ┘ ┘         │
       │ │                                      │
       │ │                 ┌ ┌ mtype     ┐ ┐    │
       │ │                 │ │ LABEL     │ │    │
       │ │ [:module-name]  │ { PROCEDURE } │    │
       │ │                 └ └ LINE      ┘ ┘    │
       │ │ MODULE                               │
       └ └ SYMBOLS                              ┘
```

Where:

DIR	displays the symbols for the current module as determined by NAMESCOPE.
DEBUG	displays the names of all debug objects.
mtype	is BIT or BYTE.
BRKREG	displays the names of all currently defined break registers.
LITERALLY	displays all currently defined LITERALLYs.
PROC	displays the names of all currently defined debug procedures.
TRCREG	displays the names of all currently defined trace registers.
PUBLICS	displays the names of public symbols.

DIR (continued)

LABEL	displays all labels in the module (with *:module-name*) or all labels with the PUBLICS attribute (with PUBLICS).
PROCEDURE	displays the names of all the procedures in the module.
:module-name	displays the symbols for the named module.
LINE	displays the line numbers of all executable statements in the *:module-name*.
MODULE	displays the names of all modules currently loaded.
SYMBOLS	displays the names of all program symbols.

Discussion

Use the DIR command to list program symbols and debug objects.

Table 3-4 compares *mytpes* with ASM-51 and PL/M-51 types.

The program data types WORD, Array, and STRUCTURE are seen as bytes by the ICE-5100 emulator. The ICE-5100 emulator only recognizes the address of the identifier that is the beginning of the array or structure. The ICE-5100 emulator does not recognize length or content information.

The 8051 object module format (OMF) specifies that arrays and structures will always be located contiguously in memory. For example, if QARRAY is located at an IDATA address in MODULE1 and contains five byte elements, then QARRAY can be accessed by specifying BYTE .:MODULE1.QARRAY for the first array element and BYTE .:MODULE1.QARRAY + 4 for the last array element. The same is true for structures; you must know the length and order of elements contained in the structure in order to access those elements.

Table 3-4 User Program Types with Corresponding *Mtype* Names

ASM-51	Corresponding *Mtype* Name
BIT	BIT
BYTE	BYTE
EQU constant	Not Supported
SET constant	Not Supported
WORD	BYTE
PL/M-51	**Corresponding *Mtype* Name**
BIT	BIT
BYTE	BYTE
WORD, ADDRESS	BYTE
STRUCTURE	BYTE
Array	BYTE

Examples

1. Display the directory of the the current module. Notice the aligned type designators and the symbol indentation indicating nesting level.

```
hlt> DIR
DIR of :MAIN_DISPLAY
   RESET_LOW . . label
   RESET_HIGH  . label
   DISP_BUFFER . <byte>
   BUFF_SIZE . . <byte>
   INT_FLAG  . . bit
   MESSAGE . . . label
   I . . . . . . <byte>
   TMO_LOW . . . <byte>
   TMO_HIGH  . . <byte>
   INIT  . . . . procedure
      TMOD . . . . . <byte>
      ETO  . . . . . bit
      EA . . . . . . bit
      TRO  . . . . . bit
   SERVICE . . . procedure
   CHAR_DISPLAY procedure
   ROTATE  . . . procedure
      INDEX_PTR  . . <byte>
      TEMP . . . . . <byte>
   START . . . . label
hlt>
```

2. Display the line numbers for MAIN_DISPLAY.

```
hlt> DIR LINE
DIR of :MAIN_DISPLAY
#1   #3   #5   #6   #7   #8   #9   #10  #11  #12
#13  #14  #15  #16  #16  #18  #19  #20  #21  #22
#23  #24  #26  #27  #28  #29  #30  #31  #32  #33
#34  #35  #36  #37  #38  #39  #40  #41
hlt>
```

3. Display the procedure names.

```
hlt> DIR PROCEDURE
DIR of :MAIN_DISPLAY
SERVICE
CHAR_DISPLAY
ROTATE
hlt>
```

DIR (continued)

4. Display all debug object names:

```
hlt> DIR DEBUG
bbb . . . . byte        01
LIT . . . . literally 'literally'
DEF . . . . literally 'define'
ccc . . . . char       'Y'
hlt>
```

Cross-Reference

DEFINE
Mtype
Symbolic Reference

Groups and executes commands

Syntax

```
DO
    [ICE-5100 emulator command(s)]
END
```

Where:

DO . . . END	is the block control keyword.
ICE-5100 emulator command(s)	is any ICE-5100 emulator command(s) except EDIT, HELP, and INCLUDE.
END[DO]	terminates the DO control block and starts execution.

Discussion

The DO . . . END block is useful inside a procedure (PROC), but can also be used to enclose several commands in a block structure. When not in a procedure definition, the DO block is executed immediately after you enter END.

Example

1. The procedure uses a DO . . . END block and accesses values stored in an array by defining a local debug object to serve as an index.

```
hlt>DEFINE PROC show = DO
.hlt>DEFINE BYTE bb
.hlt>DEFINE BYTE local_var = BYTE .:sort.currentmax
.hlt>REPEAT
..hlt>bb = BYTE .:sort.sortarray + local_var
..hlt>WRITE bb
..hlt>local_var = local_var - 1
..hlt>UNTIL local_var == 0
..hlt>ENDREPEAT
.hlt>END
hlt>END
67
34
09
08
21
hlt>
```

DO (continued)

Cross-Reference

COUNT
IF
PROC
REPEAT

DPTR

Displays or modifies the DPTR register

Syntax

DPTR [= *expression*]

Where:

DPTR	displays the contents of the DPTR (data pointer) register.
expression	evaluates to a 16-bit value.

Discussion

Use the DPTR command to display or change the contents of the DPTR register. The value is displayed or changed according to the current number BASE.

Examples

1. Display the contents of the DPTR register.

```
hlt> DPTR
012AH
hlt>
```

2. Change the contents of the DPTR register.

```
hlt> DPTR = 00FFH
hlt>
```

Cross-Reference

BASE
REGS

DYNASCOPE

Controls the operation
of NAMESCOPE

Syntax

$$\text{DYNASCOPE}\ \left[\ = \begin{Bmatrix}\text{TRUE}\\ \text{FALSE}\\ \textit{expression}\end{Bmatrix}\ \right]$$

Where:

DYNASCOPE	is a pseudo-variable that enables NAMESCOPE to be dynamic.
TRUE	turns the dynamic NAMESCOPE option on.
FALSE	turns the dynamic NAMESCOPE option off.
expression	must evaluate to TRUE (LSB = 1), or FALSE (LSB = 0).

Discussion

Normally DYNASCOPE is disabled, and NAMESCOPE remains at the address it was last assigned. Setting DYNASCOPE to TRUE enables NAMESCOPE to change whenever the current execution point changes (HALT, $, ISTEP, LSTEP or execution break).

Examples

1. Display the current setting.

```
hlt> DYNASCOPE
FALSE
hlt>
```

2. Enable NAMESCOPE to change during emulation.

```
hlt> DYNASCOPE = TRUE
hlt>
```

Cross-Reference

$
GO
HALT
ISTEP
LSTEP
NAMESCOPE
Pseudo-variable

EDIT

Invokes the debug editor

Syntax

```
 ⎧ <ESC> key
 ⎪
 ⎪        ⎡ ⎧ FILE pathname                                        ⎫ ⎤
 ⎪        ⎢ ⎪ debug-object-name                                    ⎪ ⎥
 ⎪        ⎢ ⎪ GO                                                   ⎪ ⎥
 ⎨ EDIT   ⎢ ⎨          ⎡ ⎧ OLDEST [expression] ⎫ ⎤                 ⎬ ⎥
 ⎪        ⎢ ⎪ TRACE    ⎢ ⎨ NEWEST [expression] ⎬ ⎥                 ⎪ ⎥
 ⎪        ⎢ ⎪          ⎢ ⎪ NEXT [expression]   ⎪ ⎥                 ⎪ ⎥
 ⎪        ⎢ ⎪          ⎣ ⎩ LAST [expression]   ⎭ ⎦                 ⎪ ⎥
 ⎪        ⎣ ⎩                                                      ⎭ ⎦
 ⎩
```

Where:

<ESC> key	invokes the debug editor and places the current command being entered into the editor. Pressing the <ESC> key in response to the hlt> prompt recalls the last command group for editing.
EDIT	invokes the debug editor and creates an empty edit buffer. You cannot invoke EDIT from inside a block command or debug procedure.
FILE	is a keyword that enables recalling the contents of an existing file into the edit buffer.
pathname	is the fully-qualified reference to the FILE you want to edit (e.g., :F1:MYFILE.001).
debug-object-name	recalls the named debug object definition for editing.
GO	recalls the last non-trivial GO command for editing.
TRACE	recalls the contents of the trace buffer for editing.
OLDEST	recalls the oldest instruction(s) in the trace buffer for editing. If no expression is specified, only one instruction is displayed.
NEWEST	recalls the newest instruction(s) in the trace buffer for editing. If no expression is specified, only one instruction is displayed.

EDIT (continued)

NEXT	recalls the next instruction(s) (starting with the current instruction) in the trace buffer for editing. If no expression is specified, only one instruction is displayed.
LAST	recalls the last instruction(s) (ending with the current instruction) in the trace buffer for editing. If no expression is specified, only one instruction is displayed.
expression	evaluates to an integer from 1 to +254.

Discussion

The EDIT command invokes the internal debug editor. With this editor you can create or modify previously defined debug objects, view or modify an existing system file, and view or modify the contents of the trace buffer. The *Debug Editor User's Guide*, order number 167098, shipped as part of the ICE-5100 emulator documentation, explains the menu-driven text editor.

Examples

1. Edit a debug procedure.

```
hlt> EDIT read_array
```

2. Edit the entire contents of the TRACE buffer.

```
hlt> EDIT TRACE
```

3. Edit the oldest 10 frames in the TRACE buffer.

```
hlt> EDIT TRACE OLDEST 10
```

4. Edit a file located on disk.

```
hlt> EDIT FILE testrun.log
```

Cross-Reference

Debug Editor User's Guide, order number 167098
Name
PRINT

ERROR

Controls the display of error information

Syntax

$$\text{ERROR}\left[\ = \begin{Bmatrix}\text{TRUE}\\ \text{FALSE}\\ \textit{boolean-expression}\end{Bmatrix}\right]$$

Where:

ERROR	displays the current setting (TRUE or FALSE).
TRUE	turns on the error message display; the initial setting is TRUE.
FALSE	turns off the error message display.
boolean-expression	is any expression in which the low order bit evaluates to 0 (FALSE) or 1 (TRUE).

Discussion

Set the pseudo-variable ERROR to FALSE to speed up ICE-5100 emulator operation when errors occur by eliminating the error file disk search. By entering HELP and the error number, the message can be displayed even when ERROR is set to FALSE.

Examples

1. Display the current ERROR setting.

```
hlt>ERROR
FALSE
hlt>
```

2. Enable error message display.

```
hlt>ERROR = TRUE
hlt>:MAIN_DISPLAY."PRINT
PRINT
ERROR #12
Symbol not known in current context. Symbol is either
not known, or is not local to the current module or procedure.
hlt>
```

ERROR (continued)

3. Suppress error messages by setting ERROR to FALSE and use the HELP command to display an extended error message.

```
hlt>ERROR = FALSE
hlt>:MAIN_DISPLAY."PRINT
PRINT
ERROR #12
<Error message inhibited>
hlt>HELP E12
#12
Symbol is not known in current context. Symbol is either
not known, or is not local to the current module or procedure.
<no extended help is available>
hlt>
```

Cross-Reference

HELP
ICE*nnn*
Pseudo-variable

EVAL

Calculates and displays
the result of an expression

Syntax

$$\text{EVAL } \textit{expression} \left[\begin{Bmatrix} \text{LINE} \\ \text{PROCEDURE} \\ \text{SYMBOL} \end{Bmatrix} \right]$$

Where:

EVAL	is the keyword that expands the way results of an expression are displayed.
expression	is any valid combination of values and operations.
LINE	evaluates the expression as a line number reference.
PROCEDURE	evaluates the expression as a procedure reference.
SYMBOL	evaluates the expression as a symbolic reference (label or variable).

Discussion

Use the EVAL command to display the value of expressions in all number bases and in ASCII. ASCII is displayed as a quoted string with non-printing characters indicated by periods (.). Strings longer than two characters are displayed in ASCII and as hexadecimal bytes.

When *expression* is followed by LINE, the display has the form *:module-name#line-number*. If the *expression* does not evaluate to an exact match with a line number, the system displays the line number's address that is closest to, but lower than, the value of the *expression* and adds + *offset*, the difference in bytes. The offset is displayed in the current base.

When *expression* is followed by PROCEDURE, the display has the form *:module-name.procedure-name* for only exact matches and *:module-name* for others.

When *expression* is followed by SYMBOL, the display is a fully qualified reference to the matching user symbol in all *mspaces*, with an offset for inexact matches.

EVAL (continued)

Examples

1. Display the result of a numeric calculation first without EVAL, then using EVAL. Assume the number base is hexadecimal.

```
hlt> 357T * 33H
0000471F
hlt> EVAL 357T * 33H
0100011100011111Y 18207T 471FH          '.G'
hlt>
```

2. Display the line number corresponding to an absolute address.

```
hlt> EVAL 0024H LINE
:MAIN_DISPLAY#7 + 9H
hlt>
```

4. Display the location of a procedure corresponding to a line number in a module.

```
hlt> EVAL :MAIN_DISPLAY#05 PROCEDURE
:MAIN_DISPLAY.INIT
hlt>
```

5. Display a data location as a program symbol.

```
hlt> EVAL 20H SYMBOL
(BIT)      :MAIN_DISPLAY.INT FLAG + 20H
(CODE)     <unknown>
(IDATA)    :MAIN_DISPLAY.ROTATE.TEMP + 17H
(RDATA)    <unknown>
(XDATA)    <unknown>
hlt>
```

Cross-Reference

Addr-spec
Expression
Mspace

EXIT

Terminates the debug session and returns control to the host operating system

Syntax

EXIT

Discussion

Use the EXIT command to terminate the debug session and return control to the host operating system. EXIT closes all open files.

NOTE

EXIT does not halt emulation. This allows you to leave an ICE-5100 emulator running while returning to the host operating system. When the ICE-5100 software is reinvoked, the emulation processor is reset, causing emulation to halt.

Example

1. Terminate the debug session.

```
hlt>EXIT
ICE-nnn TERMINATED
```

Expression

Numbers, variables, pseudo-variables, or functions separated by operators

Syntax

[*unary-operator*] *operand*[*binary-operator* [*unary-operator*] *operand*] [. . .]

Where:

unary-operator	acts on a single operand.
operand	can be a constant, a variable, a pseudo-variable, a function, or a sub-expression. Some operands are user-defined; others are system-defined.
binary-operator	acts on two operands. The result is a single operand.

Discussion

An expression is a combination of operands and operators. Evaluating an expression applies the operators to the operands scanning from left to right until a single result is obtained. To evaluate an expression, the system scans the expression iteratively from left to right, one iteration for each operator in the expression. An expression can be a single operand without any operators. After performing an operation, the numeric result becomes an operand for the next scan. The series of scans ends when either of two conditions occurs:

- Nothing remains except a single numeric result.
- A syntax error, type combination error, or other error occurs.

On each iteration, the scan identifies the operator that must be applied next. Table 3-5 lists operator precedence for expressions.

Table 3-5 Operators in Order of Precedence

Precedence	Operators
1	.
1	“
2	( )
3	unary + unary –
4	* / MOD
5	binary + binary –
6	NOT
7	AND OR XOR
8	NOT
8	<> >< <= >= ==

Operators

The command language contains tokens that serve as operators. Tables 3-6 and 3-7 list the unary and binary operators that the ICE-5100 emulator recognizes.

Table 3-6 Unary Operators

Operator	Operation
"	The double-quote operator enables referencing the program symbol that duplicates an ICE™-5100 emulator keyword or debug object name.
.	The dot operator returns the address of the symbolic reference.
+	Unary plus denotes a positive number.
–	Unary minus denotes a negative number and converts an unsigned value to a 2's complement signed value.
NOT	NOT is the 1's complement.

Table 3-7 Binary Operators

Operator	Function
Arithmetic	
*	Multiplication
/	Division
MOD	Modulo reduction
+	Addition
–	Subtraction
Relational	
= =	Equality
< >	Inequality
>	Greater than
<	Less than
< =	Greater than or equal to
> =	Less than or equal to
( )	For precedence or parameters
Logical	
AND	Logical operator
OR	Logical operator
XOR	Logical operator

Operands

The following sections summarize the classes of operands that the ICE-5100 emulator accepts. Within each class, some operands are user-defined and others are built-in (that is, the form is defined by the ICE-5100 emulator). The four classes of operands are constants, variables, functions, and sub-expressions.

Constants

Constants do not change value during execution. Table 3-8 summarizes the user-defined constants and boolean-constants.

An **unsigned integer constant** contains one or more valid digits and (optionally) a character indicating the number base. If you omit the base suffix, the digits are interpreted in the current base. Table 3-9 lists valid integer constants.

Table 3-8 Constants

Constant	Description
USER-DEFINED CONSTANTS	
Unsigned Integer Constants	Interpreted in the current base, stored as a word (WORD).
String Constants	ASCII characters (maximum 254), enclosed in delimiters ('). You can use one-character strings as operands with arithmetic operators.
BOOLEAN-CONSTANTS	
TRUE	Boolean value TRUE.
FALSE	Boolean value FALSE.

Table 3-9 Valid Integer Constants

Base	Valid Digits	Base Suffix
BINARY	0, 1	Y
DECIMAL	0 – 9	T
HEX	0 – 9, A – F	H

NOTE

To avoid confusion with variables and symbols, a number must not have a letter as the first digit. For this reason, a hexadecimal number requires a leading zero if the first digit is a letter (e.g., 0AB6H.)

Integers of the form *n*K are valid constants, where *n* is an unsigned decimal integer and K indicates the quantity 1024 (e.g., 4K).

A **string constant** contains up to 254 characters enclosed in apostrophes ('). To include an apostrophe in a string, use two apostrophes (' '), not a double-quote. The value of a string is its ASCII representation, with a byte for each corresponding character. You can use one-character strings as operands for arithmetic operators.

Variables

Variables store values that can change during execution or by user command. The name of the variable represents the current value. Table 3-10 summarizes the user-defined variables recognized by the ICE-5100 emulator.

User-defined variables may include all the variables listed in Table 3-10.

For example:

```
hlt> :main_display.init              /*Procedure reference*/
CODE 0F30H
hlt>
hlt> :main_display#39              /*Line number reference*/
CODE 0100H
hlt>
```

Table 3-10 User-defined Variables

Variable	Definition
Procedure reference	Returns the address of the first executable instruction in the procedure.
Line number reference	Returns the address of the first executable instruction in the line.
Label reference	Returns the address of the first executable instruction in the labeled statement.
Program variable	Returns the contents of the first byte of the data variable.
Mtype debug object	Returns the contents of the debug object.

Expression (continued)

```
hlt> :main_display.start                      /* Label reference */
CODE 008AH
hlt>
hlt> DEFINE BYTE testit = :main_display.rotate.temp
                                    /* Debug object definition */
hlt>
```

Functions

Functions can be built-in or user defined. The function returns a value to the place in the expression or command from which it was called. Functions are summarized in Table 3-11. Refer to the respective entry in this chapter for more information on a particular function.

Sub-expressions

Sub-expressions are expressions enclosed in parentheses. Parentheses override the precedence of the operators. An expression inside parentheses is evaluated first, thus becoming an operand for the rest of the expression outside the parentheses.

Examples

1. The following arithmetic expression is entered when BASE = 10t.

```
hlt> 357 + 28
385
hlt>
```

2. Logical operator with a BOOLEAN expression:

```
hlt> NOT ( (COUNT == 10) OR ($ >= 2000H) )
TRUE
hlt>
```

3. Logical operations performed on relational expressions evaluate to a Boolean value.

```
hlt> ($ == :main_display.rotate) AND (count == 5)
TRUE
hlt>
```

4. Logical operations performed on unsigned expressions return bit-by-bit values.

```
hlt> BASE = 2T
hlt> DEFINE BYTE x = 1000
hlt> DEFINE BYTE y = 1111
hlt> x AND y
0000000000001000
hlt>
```

Cross-Reference

EVAL
Mtype
Strings
WRITE

Table 3-11 Functions

Function	Description
USER-DEFINED FUNCTIONS	
Debug procedure call	A debug procedure must have a RETURN statement in its definition to be used as a function. The call then returns the Boolean expression specified in the RETURN command. (Refer to the PROC entry in this chapter.)
BUILT-IN FUNCTIONS General-purpose	
CI	Character input from the console; returns the character as the operand value.
DCI	Character input from the console; returns the character as the operand value. DCI returns a zero if no character was entered.
Strings	
CONCAT (*'string-ref'*[, . . .])	Creates a new string by concatenation.
INSTR (*'string1'*, *'string2'*)	Searches for *'string2'* within *'string1'* and returns the index of the first character.
NUMTOSTR (*expression*)	Converts the expression into ASCII.
STRLEN (*'string-ref'*)	Returns the number of characters in the string.
STRTONUM (*'string'*)	Returns the numeric value of the ASCII string.
SUBSTR (*'string-ref'*,*start*, *length*)	Returns a substring starting at the character indexed by *start*.

GO

Starts program execution and controls break and trace functions

Syntax

```
                           ┌ ┌ ┌ FOREVER                                   ┐ ┐ ┐
                           │ │ │                                           │ │ │
                           │ │ │ USING { BRKREG                  }         │ │ │
   GO ┌ FROM addr-spec  ┐  │ │ │       { brkreg-name[, . . .]  }         │ │ │
      └ ARM [addr-spec] ┘  │ │ │                                           │ │ │
                           │ └ └ TIL [OUTSIDE][PAGE] addr-spec [, . . .]   ┘ ┘ │
                           │                                                   │
                           │                 { trcreg-name                   } │
                           │        TRACE    { [FROM addr-spec][TIL addr-spec] } │
                           └                 { [OUTSIDE][PAGE] address       } ┘
```

Where:

GO	starts program execution from the current execution point with all existing break and trace specifications. The ICE-5100 emulator's initial condition is to GO FOREVER.
FROM	changes the current execution point to the *addr-spec* specified. The *addr-spec* must be the boundary of a machine language instruction.
ARM	specifies that all breakpoints are to be enabled upon execution of the specified *addr-spec*. ARM without an *addr-spec* clears any previous ARM condition.
addr-spec	is a memory location or range of locations.
FOREVER	clears all breakpoints set by GO TIL or GO USING and starts program execution. The ICE-5100 emulator retains previous breakpoints until you execute a GO TIL, GO USING, GO FOREVER, or RESET ICE command. GO FOREVER is the initial default condition.
USING	specifies that a break register (BRKREG) will be used to set breakpoints.
BRKREG	specifies that all defined break registers will be used during program execution. If the combined total of all breakpoint types exceeds four, the ICE-5100 emulator will not enter emulation.
brkreg-name	is the name of a previously defined break register (BRKREG).
TIL	signals that one or more breakpoints will follow.

OUTSIDE	specifies that the ICE-5100 emulator is to gather trace information for all addresses other than those specified in the *addr-spec*.
PAGE	specifies a range of addresses for trace information that is exactly 256 bytes in length. A page is specified with a single 8-bit address. Valid page addresses range from 00H to 0FFH with PAGE 00H beginning at address 0000H. You can also specify a range of pages.
TRACE	is used to limit collection of trace information or to signal that the limits are in a trace register. With no TRACE options, all trace information is collected during emulation.
trcreg-name	is the name of a previously defined trace register (TRCREG).

Discussion

The GO command begins execution of the user's program. Execution starts from the current execution point. The execution point may be changed by assigning a value to the program counter ($) or by using the FROM option.

NOTE

Changing the current execution point ($) will cause trace data collection to begin with an empty trace buffer when a GO command is executed.

Emulation of the program is indicated by the emu> prompt. Once execution is started with the GO command, it continues until a breakpoint is encountered or until a HALT command is entered. Edit the most recent GO command by entering EDIT GO.

You can enter any ICE-5100 emulator command during emulation *except* for the following:

$	ADDRESS	ASM	BIT
BOOLEAN	BYTE	CHAR	EDIT TRACE
GO	INT	ISTEP	LOAD
LSTEP	MAP	PRINT	REGS
RESET	CHIP	R0-R7	SAVE
VERSION	WORD		

The following sections explain when to choose one form of the GO command over another. The types of conditions recognized by the ICE-5100 emulator are break specifications and trace specifications.

Execution Without Breakpoints

To start program execution without setting breakpoints, use the FOREVER option (initial condition). Stop execution of the program by entering HALT. Resume emulation from the current execution point by entering GO.

Execution With Breakpoints

To conditionally break execution, use the GO TIL or the GO USING options. The TIL option lets you specify break conditions directly on the GO command line. Use the TIL option for simple break conditions which will be used only once.

The USING option requires defined BRKREG(s). Use the USING option with the BRKREG keyword or a break register name when the break condition exceeds one line or when the break condition will be used more than once.

Break registers are debug objects that can be saved to a file and used again in future sessions as well as used again before exiting the present debug session.

The ICE-5100 emulator supports three types of breakpoints as described below:

- Specific break - Emulation halts when a specific instruction address is executed.
- Range break - Emulation halts when an instruction address within a specified range of addresses is executed. A range break counts as three specific break specifications.
- Page break - Emulation halts when an instruction address within a page (256 bytes) is executed. Any number of page breaks count as one specific break.

The ICE-5100 emulator keeps track of the break types used and issues an error message if a GO command is entered with more break options than the hardware can support. Up to four specific breaks per GO command are accepted by the ICE-5100 emulator.

Collection of Trace Data

The initial condition is for trace information to always be collected. Use the TRACE option with the GO command to limit trace collection.

Using the TRACE FROM and TIL options causes tracing to be enabled at the FROM *addr-spec* and disabled at the TIL *addr-spec*. When tracing is specified with an *addr-spec*, tracing is conditionally enabled within the specified *addr-spec*. Trace conditions can be saved in a TRCREG, and the name of the TRCREG can be included with the TRACE option.

NOTE

The instruction which causes an emulation break is always collected in the trace buffer regardless of the trace specification.

The ICE-5100 emulator supports three types of tracepoints as described below:

- Specific trace - Traces a specific instruction address.
- Range trace - Traces instruction addresses within a specified range of addresses.
- Page trace - Traces instruction addresses within a page (256 bytes)

The ICE-5100 emulator accepts only one type of tracepoint per GO command.

Defining an ARM Breakpoint Condition

Initially, all specified breakpoints are enabled when you enter emulation without an ARM specification. If an ARM specification was used in a previous GO command, then that ARM specification (whether met or not) will remain in effect until the ARM specification is respecified or disarmed.

Use the ARM option without the *addr-spec* to remove (disarm) the ARM condition. Issuing a GO command with an ARM specification does not remove any previously defined break or trace points.

Examples

1. The following example shows a simple GO command. The ICE-5100 emulator starts execution from the address specified in the program counter register.

```
hlt> GO
emu>
```

2. Specify a starting address in the GO command. Any existing breakpoints are cleared.

```
hlt> GO FROM 05H FOREVER
emu>
```

3. Stop execution at a line number.

```
hlt> GO TIL :MAIN_DISPLAY#24
emu>
Probe stopped at 0080H (:MAIN_DISPLAY#27) because of
internal break.
```

GO (continued)

4. Use a break register to stop execution at line #32.

```
hlt> DEFINE BRKREG stop = :MAIN_DISPLAY#32
hlt> GO USING stop
emu>
Probe stopped at 0080H (:MAIN_DISPLAY#32 + 2H) because of
internal break.
 Break Register is [STOP]
hlt>
```

5. Execute a program until it is outside a page.

```
hlt> GO FROM 0 TIL OUTSIDE PAGE 1
emu>
```

6. The following examples show possible tracing options.

```
hlt> GO FROM 0 TRACE trc1        /* Trace using trace register */
emu>

hlt> GO FROM 0 TRACE 16H TO 43H
emu>
hlt> GO FROM 0 TRACE 10H LENGTH 20H
emu>

hlt> GO FROM 0 USING BRKREG TRACE FROM 0 TIL 17H
emu>
```

Cross-Reference

Addr-spec
BRKREG
DEFINE
Name
TRCREG

HALT

Halts emulation

Syntax

HALT

Discussion

The HALT command halts emulation. HALT has no effect on breakpoints or tracepoints. Use the GO command to resume program execution.

NOTE

If you are using a BRKREG with the CALL *proc* option, you will not be able to enter HALT. In this case, enter <CNTL>C (<CNTL><BREAK> on IBM hosts) to abort the debug procedure.

Example

1. Halt program execution.

```
hlt> GO FOREVER
emu> HALT
Probe stopped at 008CH(:MAIN_DISPLAY#37) because of user halt.
hlt>
```

HELP

Provides on-line help

Syntax

```
       ┌ name ┐
       │ En   │
HELP   │ E    │
       │ n    │
       └      ┘
```

Where:

HELP	displays the list of keywords for which HELP is available.
name	is one of the help topics displayed by HELP. Entering HELP *name* displays information for that topic.
E*n*	displays the extended error message for error *n*. The existence of extended messages is indicated by a [∗] symbol following the message. The *n* must be a decimal number.
E	requests the extended error message for the last error.
n	displays error message for error *n* without any extended text. The *n* must be a decimal number.

Discussion

The HELP topic gives you access to on-line information about ICE-5100 emulator topics and error messages. Type HELP followed by an ICE-5100 emulator topic and a summary of the topic will be displayed on the screen.

If an error message occurs that has an asterisk enclosed in brackets [∗] next to it, use the E*n* option to get extended information about the error.

You cannot use the HELP command within a block structure (REPEAT, DO/END, COUNT, IF), or within a debug procedure (PROC).

Examples

1. Display the list of available help information. (Listing is for the ICE-5100/252 user probe.)

```
hlt>HELP

HELP is available for:

ADDRESS      APPEND       ASM        BASE        BIT          BOOLEAN
BRKREG       BYTE         CHAR       CI          CNTL_C       COMMENTS
CONSTRUCTS   COUNT        CPU        CURHOME     CURX         CURY
DCI          DEBUG        DEFINE     DIR         DISPLAY      DO
DYNASCOPE    EDIT         ERROR      EVAL        EXIT         EXPRESSION
GO           HELP         IF         INCLUDE     INVOCATION   ISTEP
KEYS         LABEL        LINES      LIST        LITERALLY    LOAD
LSTEP        MAP          MENU       MODIFY      MODULE       MSPACE
MTYPE        NAMESCOPE    OPERATOR   PAGING      PARTITION    PRINT
PROC         PSEUDO_VAR   PUT        REFERENCE   REGS         REMOVE
REPEAT       RESET        RETURN     SAVE        STRING       SYMBOLIC
SYNCSTART    TEMPCHECK    TRCREG     TYPES       VARIABLE     VERIFY
VERSION      WAIT         WORD       WRITE
```

2. Display HELP for EXIT.

```
hlt>HELP EXIT
The EXIT command terminates a debug session.
hlt>
```

3. Display extended help for an error number.

```
hlt>HELP E10
#10
Cannot determine current default module. [*]
Could not find current location in any known module. The current
execution point is either outside of the program or in a module
for which there is no symbol information.
hlt>
```

ICE*nnn*

Invokes the ICE-5100/*nnn* user probe software

Syntax

```
                                 ┌ CRT [ (filename) ] | NOCRT          ┐
                                 │ MACRO [ (filename) ] | NOMACRO       │
                                 │ SUBMIT | NOSUBMIT                    │
                                 │ CFG [ (filename) ] | NOCFG           │
                                 │ ERROR (filename)                     │
[RUN][pathname]ICEnnn            │ HELP (filename)                      │ [...]
                                 │ VSTBUFFER (number)                   │
                                 │ BAUD (baud-rate)                     │
                                 │ CHANNEL (number)                     │
                                 └ Pnnn (filename)                      ┘
```

Where:

RUN	is a Series III utility that invokes the 8086 processor instead of the default 8085 processor.
pathname	is the device and/or directory where the ICE software is filed.
ICE*nnn*	loads the ICE-5100/*nnn* emulator software.
nnn	is replaced by the number of your user probe, (e.g., 252).
CRT	sets CRT parameters from the specified file or from the ICE*nnn*.CRT file if no file was specified. NOCRT prevents loading the ICE*nnn*.CRT file. Abbreviations are CR and NOCR. CRT is the default.
MACRO	invokes ICE-5100 emulator commands from the specified file or from the ICE*nnn*.MAC file if no file was specified. The commands are executed during software initialization. NOMACRO prevents the ICE-5100 emulator from loading the ICE*nnn*.MAC file. Valid abbreviations are MR and NOMR.
SUBMIT	indicates the ICE-5100 emulator will be invoked from within an ISIS SUBMIT or DOS batch file. NOSUBMIT is the default. Valid abbreviations are SM and NOSM.
CFG	sets invocation controls from the specified file or from the ICE*nnn*.CFG file if no file was specified. NOCFG prevents loading the ICE*nnn*.CFG file. CFG is the default.
ERROR (*filename*)	selects the ICE-5100 emulator's error text file.
HELP (*filename*)	selects the ICE-5100 emulator's help text file.

VSTBUFFER (*number*)	increases the size of the the virtual symbol table buffer. VSTB is the abbreviation. The *number* can be from 5 KB to 61KB.
BAUD (*baud-rate*)	selects a baud rate. Valid rates are 300, 1200, 9600 (default for IBM hosts), and 19200 (default for Intel hosts).
CHANNEL (*number*)	specifies the serial channel of the host system. *Number* can be 1 (default channel) or 2. CH is the abbreviation.
P*nnn* (*filename*)	selects the ICE-5100 emulator's probe software file.

Discussion

The ICE*nnn* command invokes the ICE-5100 emulator software from the host operating system. You cannot specify the command once the ICE-5100 emulator software is running.

Examples

1. The following example invokes the software for the ICE-5100/252 user probe from a Series III.

```
-RUN ICE252 MACRO (loader.mac) BAUD (9600) VSTB (12)
```

2. The following example invokes the software for the ICE-5100/252 user probe from an IBM PC.

```
C:\>ICE252 MACRO (loader.mac) BAUD (19200) CH (2) VSTB (12)
```

3. The following example invokes the software for the ICE-5100/252 user probe from a Series IV.

```
>ICE252 MACRO (loader.mac) VSTB (12)
```

The following sections contains information on how or when to use the invocation options.

CRT(*filename*) | NOCRT

The ICE-5100 emulator is designed to run on an Intel Series III/IV, IBM PC AT, or PC XT computer system. The CRT configuration file is only necessary when a non-standard terminal is used with the ICE-5100 emulator. A CRT configuration file contains control code definitions to enable the CRT based software of the ICE-5100 emulator to function properly. Create a CRT file only if you wish to use a terminal other than an Intel or IBM terminal.

If you name a CRT file ICE*nnn*.CRT and locate it on the same device or directory as ICE*nnn*.86 (or ICE*nnn*.EXE), the CRT file is automatically opened when the software is invoked. It is not necessary to specify CRT on the invocation line, but the terminal is configured whenever ICE-5100 emulator software is invoked. To suppress the CRT file, invoke the software as follows:

```
ICEnnn NOCRT
```

If the CRT file is named something other than ICE*nnn*.CRT, include the CRT option when invoking ICE*nnn*. The following example includes a CRT file named *termnl.crt*.

```
ICEnnn CRT(termnl.crt)
```

To create a CRT file for a non-standard terminal, use the terminal's instruction manual and compare requirements with the codes the ICE-5100 emulator uses for terminal configuration. A CRT file is a text file containing alteration commands. Alteration commands modify the environment and the communication link between the ICE-5100 emulator software, the keyboard, and the screen. Table 3-13 and Table 3-14 explain alteration commands.

Alter Environment Commands change the way data is displayed on the screen. To enter a command, precede the code with an "A".

Alter Function Commands specify the function codes for control characters and cursor control. To enter alter-function values, precede the code with AF, then enter an equal sign and a value. Table 3-14 contains the alter-function codes and explains the values.

Table 3-12 Intel Terminal Control Codes

Cursor Function	Code Produced	Function Code	Code Sent to CRT
Down	1CH,0	CD	1BH,42H
Home	1DH,0	CH	1BH,48H
Left	1FH,0	CL	1BH,44H
Right	14H,0	CR	1BH,43H
Up	1EH,0	CU	1BH,41H
CRT Function	**Function Code**	**Code Sent to CRT**	
Clear all of screen	ES	1BH,45H	
Clear rest of screen	ER	1BH,4AH	
Clear rest of line	EL	unavailable	
Clear line	EK	1BH,4BH	
Blankout character	BK	20H (ascii blank)	

Examples

1. The first example shows the format for alter-environment commands.

```
AV=22    /*changes the number of lines on the screen to 22.*/

AB=7  /*changes the keyboard BREAK character to <CNTL><G>.*/
```

2. This example shows the format for alter-function commands.

```
AFMD=0A        /*specifies that the code required to move the cursor down the
                screen is a single linefeed (0AH).*/

AFER=        /*specifies that the terminal being used does not have an "erase
              rest of screen" code.*/
```

Table 3-13 Alter-Environment Commands

Code	Default	Value
B	1BH	ESCAPE. A new value should be assigned for terminals that require the <ESC> key for control sequences.
O*	0*	OFFSET value to be added to the row and column numbers following the cursor control sequence of "alter function command" (AFAC). Enter as a single hexadecimal byte.
R	7FH	RUBOUT. This key moves the cursor one character to the left and deletes that character.
T	8	TAB. The value can be any integer from 0 to 79 decimal. The default setting is 8. TABS occur every 8 spaces. Enter as a decimal integer.
V	25	Number of screen lines. The possible values are 22, 23, 24, or 25. Text area for ICE-5100 emulator is 20 lines. Enter value as a decimal integer.
W	F	Wrapping. T(rue) indicates the terminal will generate a *carriage-return* and *linefeed* if a character is printed in column 80. F(alse) indicates a non-wrapping terminal.
X	T	Cursor format. The format is an ordered pair of x, y coordinates. T sets the format as column, row. F sets the format as row, column. Used with AFAC.

*The code is the letter O, and the default is the number zero.

Table 3-14 Alter-Function Values

Code	Default	Value
AC	00H	Cursor movement command used by the terminal. The coordinates of the cursor address follow the code. The coordinates are given in the order specified by the AX command, and the offset specified by the AO command.
BK	20H	Code that blanks out a single screen location.
CD	1CH	Cursor down code.
CH	1DH	Cursor home code.
CL	1FH	Cursor left code.
CR	14H	Cursor right code
CU	1EH	Cursor up code.
DL	null	Delete line code. Used to speed up display on the Hazeltine 1510 and similar terminals.
EK	1BH,4BH	Code to erase the entire line.
EL	null	Code to erase the line following the cursor.
ER	1BH,4AH	Code to erase the screen following the cursor.
ES	1BH,45H	Code to erase the entire screen.
IG	null	Ignore the byte whenever it is received as keyboard input. If IG is set to 00H, all bytes are accepted. IG code is needed for terminals that have multiple character codes for UP and DOWN, such as the Hazeltine 1510. IG should be set to the lead-in (right-curly brace). UP and DOWN should be set to the second letter of the cursor up and down key code. This avoids problems if there is no type ahead buffer.
IL	null	Insert line code. Used for reverse scrolling.
MB	0DH	Code to move the cursor to the beginning of the line.
MD	1BH,42H	Code to move the cursor down.
MH	1BH,48H	Code to move the cursor to the home position.
ML	1BH,44H	Code to move the cursor to the left.
MU	1BH,41H	Code to move the cursor up.
XA	01H	<CNTL>A
XF	06H	<CNTL>F
XX	18H	<CNTL>X
XZ	1AH	<CNTL>Z

3. The following example shows a CRT text file. More than one command is allowed on a line when separated with a semi-colon (;), otherwise commands are separated by a carriage return.

```
AV=24;AB=1B;AR=7F;AFBK=20
AFXA=1;AFXF=6;AFXU=15;AFXX=18;AFXZ=1A
AFCD=16;AFCH=1E;AFCL=08;AFCR=0C;AFCU=0B
IFMB=0D;AFMD=1B59;AFMH=1E;AFML=08;AFMR=0C;AFMU=0B
AFES=1B2B;AFER=1B59;AFEK=;AFDL=1B52;AFEL=1B54
AFIL=1B45;AFAC=1B3D;A0=20;AX=F
```

MACRO(*filename*) | NOMACRO

A macro file is useful for customizing the debugging environment. An ICE*nnn* macro consists of ICE-5100 emulator commands that can be activated when the software is invoked. If you name a macro file ICE*nnn*.MAC and locate it in the same directory as ICE*nnn*.86 or ICE*nnn*.EXE, the macro file is automatically opened when the ICE-5100/*nnn* software is invoked. To suppress a MAC file, invoke the software as follows:

```
ICEnnn NOMR
```

If the macro file is named something other than ICE*nnn*.MAC, include the MACRO option with the macro filename when invoking the ICE-5100 emulator software. The following example includes a MACRO file.

```
ICEnnn MR debug.mac
```

Note: If at any time you want to include a macro file without reinvoking the software, you can use the INCLUDE command to load the macro file.

Example

1. The following example macro file defines literallys and uses several ICE-5100 emulator commands to customize a debugging session.

```
BASE = HEX
MAP ICE
LOAD test.pro
DEFINE LITERALLY def = 'DEFINE'
def LITERALLY lit = 'LITERALLY'
def lit h = 'HALT'
def lit ef = 'EDIT FILE'
def lit rc = 'RESET CHIP'
def lit bc = 'BYTE CODE'
def lit br = 'BYTE RDATA'
def lit bx = 'BYTE XDATA'
def lit pa = 'PRINT ALL'
def lit pn = 'PRINT NEWEST'
def lit brk = 'BRKREG'
def lit trc = 'TRCREG'
def lit dbr = 'DEFINE BRKREG'
def lit dtr = 'DEFINE TRCREG'
def lit g = 'GO'
def lit g0 = 'GO FROM 0'
def lit I = 'ISTEP'
def lit gu = 'GO USING'
def lit rep = 'REPEAT'
def lit r = 'REGS'
def lit inc = 'INCLUDE'
def lit nl = 'NOLIST'
inc debug.tst nl
```

SUBMIT | NOSUBMIT

When SUBMIT is entered as an option, the ICE-5100 emulator software will execute all further emulator commands from an ISIS SUBMIT or DOS BATCH file. Typically, these commands are stored in an executable ISIS.csd or DOS.bat file. Check documentation for your host operating system, and refer to the software chapters in the *ICE™-5100 Emulator Installation Supplement*, order number 167095.

CFG (*filename*) | NOCFG

A configuration file is used to store ICE-5100 emulator invocation controls. If you name a file ICE*nnn*.CFG and locate in the same directory as ICE*nnn*.86 or ICE*nnn*.EXE, the configuration file is automatically opened when the emulator software is invoked.

If there are conflicts between controls in the configuration file and the invocation line, the ICE-5100 emulator uses the information on the invocation line. To suppress the ICE*nnn*.CFG file, invoke the software as follows:

```
ICEnnn NOCFG
```

The configuration file is a text file; it uses the same syntax as the invocation line. The following example is a typical configuration file.

```
CRT (mycrt.CRT) MACRO (mymac.MAC) VST (10) CH (2)
```

If the configuration file is named something other than ICE*nnn*.CFG, include the CFG option with the configuration filename when invoking the ICE-5100 emulator software. The following example opens a configuration file named *debug.cfg*.

```
ICEnnn CFG (debug.cfg)
```

HELP(*filename*)
ERROR(*filename*)

The default HELP and ERROR files are ICE*nnn*.OVH and ICE*nnn*.OVE. The ICE-5100 emulator assumes that these files are in the same directory as ICE*nnn*.86 or ICE*nnn*.EXE. If they are located elsewhere, a path to the files must be included upon invocation or no HELP or ERROR messages will be available when the ICE-5100 emulator software is run.

VSTBUFFER(*number*)

The VSTB(*number*) option specifies the in-memory size of the virtual symbol table buffer. The *number* is a positive integer representing memory size in kilobytes. Increase the size of the virtual symbol table buffer to reduce symbol table manipulation time.

The minimum allowable size of the virtual symbol table buffer is 5 KB. The default size is 5 KB. The maximum size of the virtual symbol table buffer is 61 KB.

The larger the resident portion of the virtual symbol table, the less time the ICE-5100 emulator spends accessing the virtual symbol table. Increasing the buffer size uses more memory but improves performance. As a rule of thumb, a VST buffer size equal to one-half the size of the absolute object file will usually yield the best performance.

VSTBUFFER Examples

1. Increase the size of the virtual symbol table buffer.

```
ICEnnn VSTB(21)
```

2. Decrease the size of the virtual symbol table buffer by exiting and re-invoking the software with another option.

```
EXIT
ICE-5100/252 terminated
ICEnnn VSTBUFFER(12)
```

BAUD(*baud-rate*) CHANNEL(*number*)

At invocation, select the baud rate and serial communication channel for the host operating system when you want a rate and channel other than the default. The baud rate and serial channel can also be changed with the RESET ICE command after invocation.

P*nnn* (*filename*)

The probe software file is ICE*nnn.nnn*. The ICE-5100 emulator assumes this file is in the same directory as ICE*nnn*.86 or ICE*nnn*.EXE. If the file is located elsewhere, a path to the file must be included upon invocation. For example:

```
ICEnnn Pnnn(filename)
```

IF

Groups and conditionally
executes commands

Syntax

IF *boolean-condition* THEN

[*ICE-5100 emulator command(s)*]

[ELSE [*ICE-5100 emulator command(s)*]]

END[IF]

Where:

IF	is the conditional control keyword.
condition	is an expression that evaluates to TRUE or FALSE.
THEN	signals that the following commands will be executed if the Boolean condition is TRUE.
ELSE	is an optional clause. Commands that follow will be executed if the Boolean condition is FALSE.
commands	are one or more ICE-5100 emulator commands except EDIT, HELP, and INCLUDE.
END[IF]	terminates the IF block.

Discussion

An IF block is executed immediately after you enter its END statement.

IF (continued)

Example

1. Create a debug procedure containing an IF block. The debug procedure returns TRUE if the number passed as a parameter is evenly divisible by three. Note the command nesting caused by IF, indicated by the .hlt> prompt.

```
hlt> DEFINE PROC divthree = DO
.hlt> IF (%0 mod 3) == 0 THEN
..hlt> RETURN TRUE
..hlt> ELSE
..hlt> RETURN FALSE
..hlt> ENDIF
.hlt> END
hlt> divthree (6)
TRUE
hlt> divthree (5)
FALSE
hlt>
```

Cross-Reference

PROC

INCLUDE

Retrieves command definitions
from a disk file

Syntax

INCLUDE *pathname* [NOLIST]

Where:

INCLUDE	loads a command file into memory.
pathname	is the fully-qualified reference to the INCLUDE file (e.g., WD0/ICEDIR/LOADER.MAC).
NOLIST	suppresses the listing of the included file to the screen.

Discussion

The INCLUDE command retrieves a command file. Command files are created in two ways: create a file (such as a macro) using the debug editor or store definitions created during a debug session in a file with the PUT or APPEND command. After a file is included, activate individual debug objects by referring to them by name. Use DIR DEBUG to get a listing of active debug objects.

INCLUDE has the following restrictions:

- Nesting of INCLUDE commands is limited by available memory.
- An INCLUDE command cannot appear in block structures (i.e., REPEAT, COUNT, IF, DO/END, or a debug procedure).
- The INCLUDE command must be the last command on a line.

Examples

1. Retrieve a debug procedure from a file.

```
hlt>INCLUDE C:\SCRAPDIR\
hlt>DEFINE PROC test_tank = DO
.hlt>IF (:node2.tank status <> 0) THEN
..hlt>WRITE 'Status of tank is ', :node2.tank_status
..hlt>ELSE
..hlt>WRITE 'Tank not in use'
..hlt>ENDIF
.hlt>END
hlt>
```

2. Use the NOLIST option to suppress the file contents display to the screen.

```
hlt> INCLUDE C:\SCRAPDIR\ NOLIST
hlt>
```

Cross-Reference

APPEND
DIR
EDIT
Pathname
PUT

INSTR

Returns the index of a substring within a given string

Syntax

INSTR (*string-ref1*, *string-ref2*[, *start*])

Where:

string-ref	can be characters enclosed in apostrophes, a string expression using CONCAT, NUMTOSTR, or SUBSTR functions, or a reference to a CHAR type debug object. Use the continuation character (&) when a command is longer than one line.
start	defines where to begin the search in *string-ref1*. The value *start* is an index number or an expression that evaluates to an index number from 1 through 254 in the current number base.

Discussion

INSTR searches for *string-ref2* within *string-ref1* and returns the index (in decimal) of the first character of *string-ref2*. If *start* is larger than *string-ref-1*, if *string-ref-2* is not found in *string-ref-1*, or if *string-ref-1* is null, the ICE-5100 emulator returns zero.

Examples

1. Return the decimal (BASE = 10t) index of the first occurrence of the substring 'def'.

```
hlt> INSTR ('abcdef', 'def')
4
hlt>
```

2. Define a string variable "longmsg" find the index of the first instance of the substring 'Add', and locate the second substring 'Add' by including an index of 10.

```
hlt> DEFINE CHAR longmsg = 'Addresses < 0 are invalid. ' &
hlt>> 'Addresses >64K are also invalid.'
hlt> INSTR (longmsg, 'Add')
1
hlt> INSTR (longmsg, 'Add', 10)
29
hlt>
```

INT

Displays interrupts in progress

Syntax

INT

Discussion

The INTerrupt command displays interrupts in progress or pending when emulation was halted. If no interrupts were in progress, the word NONE is displayed.

Clear any interrupts in progress by executing a RETI for each interrupt or by entering a RESET CHIP command.

Example

1. Display the interrupts that were in progress when emulation was halted.

```
hlt> INT
Interrupts In Progress
----------------------
Level 0 -- Timer 1
Level 1 -- NONE
hlt>
```

Cross-Reference

Expression

ISTEP

Single-steps through user programs by machine language instructions

Syntax

ISTEP [*number*] [FROM *addr-spec*]

Where:

ISTEP	executes a machine language instruction.
number	is an unsigned integer expression in the current base specifying the number of steps to take. The default increment is 1. The maximum increment value is 255T.
FROM	specifies a starting address where ISTEPs are to begin. The default start address is the current execution point ($).
addr-spec	is a memory location in CODE.

Discussion

The ISTEP (instruction step) command single-steps through user programs by machine language instructions. An ISTEP command executes one instruction and halts. A break message specifying the current execution point ($) and opcode disassembly is displayed when ISTEP halts.

Instructions executed via the ISTEP comand are entered into the trace buffer regardless of the last trace option entered. Using the FROM option will clear the trace buffer as does any operation that changes the current execution point ($).

Example

1. Step and display the next instruction.

```
hlt> ISTEP FROM 300AH
300CH 758108      MOV     SP,#08H
hlt>
```

2. Step through a range of instructions.

```
hlt> $ = 0
hlt> ISTEP 3
00E9H 02008A      LJMP          (:MAIN_DISPLAY#01)
hlt>
```

Cross-Reference

$
Addr-spec
Expression
LSTEP
Mspace
PRINT

Keywords

Words reserved for use by the ICE-5100 emulator

The ICE-5100 emulator uses the following words and symbols. Do not define a debug object to have the same name as a keyword. If your program has a variable or procedure name that duplicates a keyword, use the quote (") operator when you reference that variable or procedure name (for example, "EXIT).

NOTE

The following list does not include any words or symbols reserved for your user probe. Refer to your user probe supplement for additional keywords.

"	$	%	(	)	*
+	,	–	.	/	:
;	<	< =	< >	=	= =
>	> =	\	ADDRESS	ALL	AND
APPEND	ARM	ASM	BASE	BAUD	BINARY
BIT	BOOLEAN	BREAK	BRKREG	BYTE	CALL
CAUSE	CHANNEL	CHAR	CHIP	CI	CLEAR
CLEAREOL	CLEAREOS	CLIPS	CODE	CONCAT	COUNT
CPU	CURHOME	CURX	CURY	DEBUG	DECIMAL
DEFINE	DIR	DISARM	DO	EDIT	ELSE
END	ENDCOUNT	ENDIF	ENDREPEAT	ERROR	EVAL
EXIT	FALSE	FILE	FOREVER	FROM	GLOBAL
GO	HALT	HELP	HEX	ICE	IDATA
IF	INCLUDE	INPUT	INSTR	ISTEP	INT
LABEL	LAST	LENGTH	LINE	LIST	LITERALLY
LOAD	LSTEP	MAP	MENU	MOD	MODULE
NAMESCOPE	NEWEST	NEXT	NOCLIPS	NOCODE	NOLINES
NOLIST	NOSYMBOLS	NOT	NP	NUMTOSTR	OLDEST
OR	OUTPUT	OUTSIDE	PAGE	PRINT	PROC
PROCEDURE	PUBLICS	PUT	RBANK	RDATA	READ
REGS	REMOVE	REPEAT	RESET	RETURN	SAVE
SCREEN	STRLEN	STRTONUM	SUBSTR	SYMBOL	SYMBOLIC
SYMBOLS	SYNCSTART	TEMPCHECK	THEN	TIL	TO
TRACE	TRCREG	TRUE	UNTIL	USER	USING
VERIFY	VERSION	WAIT	WHILE	WORD	WRITE
XDATA	XOR				

LIST/NOLIST

Opens or closes a list file

Syntax

$$\left\{ \begin{array}{l} \text{LIST} \left\{ \begin{array}{l} \textit{pathname} \\ ? \end{array} \right\} \\ \\ \text{NOLIST} \end{array} \right\}$$

Where:

LIST ?	displays the name of the currently open list file, if any.
LIST *pathname*	opens a list file named *pathname*.
pathname	is the fully-qualified reference to the list file. This can include references to line printers, e.g., LIST:LP:.
NOLIST	closes the open list file.

Discussion

Typically, a list file is used as a debug session log. After specifing LIST *pathname*, all interactions between the ICE-5100 emulator and the terminal (except edits) are recorded in a permanent file. You can open only one list file at a time. Close list files with the NOLIST command, by opening another LIST file, or with the EXIT command.

If a file already exists, the message "Overwrite existing file? (y or [n])" is displayed. A response other than 'y' or 'Y' aborts the LIST. A response of 'y' or 'Y' causes the contents of the existing file to be overwritten.

Examples

1. Open a list file, then display the pathname.

```
hlt>LIST A:JUNE16.86
hlt>LIST ?
<List file is A:JUNE16.86>
hlt>
```

2. Close the list file.

```
hlt>NOLIST
hlt>
```

3. Open a list file to a line printer.

```
hlt>LIST :LP:          /* From a Series III or Series IV host */
```

or

```
hlt>LIST PRN:                              /* From an IBM PC host */
```

Cross-Reference

Pathname

LITERALLY

Is a debug object that abbreviates or renames a command or string

Syntax (two forms)

1. Define a LITERALLY:

   DEFINE LITERALLY *name* [= '*character-string*']

2. Display a LITERALLY:

   LITERALLY *name*

Where:

DEFINE	defines or redefines a debug object.
LITERALLY	indicates a string replacement.
name	is the alias for the replacement string which follows.
'*character-string*'	is an ASCII string which can be up to 254 characters in length.

Discussion

LITERALLY definitions are special debug objects with character strings as values. With LITERALLY definitions, you can abbreviate keywords or complete commands.

LITERALLY definitions are always global and must not duplicate a keyword.

When scanning for command input, the ICE-5100 emulator recognizes and replaces LITERALLYs with their character string definitions. If MENU = FALSE, the LITERALLY is not expanded on the screen, but is interpreted correctly as its defined character string.

Use the EDIT command to modify LITERALLY definitions and the REMOVE command to delete LITERALLY definitions. LITERALLY definitions can be saved to a file with the PUT or APPEND command, and included in future debug sessions with the INCLUDE command. The DIR LITERALLY command lists the names of all LITERALLY objects.

Examples

1. Create a new LITERALLY definition.

```
hlt>DEFINE LITERALLY def = 'define'
hlt>
```

2. Display a LITERALLY definition.

```
hlt> LITERALLY len
DEFINE LITERALLY len ='length'
hlt>
```

3. Display the directory of LITERALLY definitions.

```
hlt> DIR LITERALLY
DEF . . . 'define'
LIT . . . 'literally'
LEN . . . 'length'
BC  . . . 'byte code'
PA  . . . 'print all'
RC  . . . 'reset chip'
hlt>
```

4. Delete the LITERALLY definition for LEN.

```
hlt> REMOVE len
hlt>
```

Cross-Reference

APPEND
DEFINE
DIR
EDIT
Name
PUT
REMOVE

LOAD

Loads executable code from a file into mapped program memory

Syntax

LOAD *pathname* [NOCODE] [NOLINES] [NOSYMBOLS] [APPEND]

Where:

LOAD	loads the designated object file.
pathname	is the fully-qualified reference to the object file you want to load into program memory.
NOCODE	ignores content records in the object file. NOTE: Loading user code with the NOCODE option disallows use of the NAMESCOPE feature.
NOLINES	ignores debug line number information when loading.
NOSYMBOLS	ignores debug symbol and line number information when loading.
APPEND	allows multiple symbolic loads without purging the symbol table.

Discussion

Before loading, you must compile or assemble your program using the DEBUG option and link your program. The ICE-5100 emulator only provides full symbolic support if the object file is RL51-generated and contains all available debug information. The NODEBUGLINES option must *not* be specified during the RL51 process in order for NAMESCOPE to work properly with PL/M-51 modules.

The LOAD command options enable you to restrict the amount of information that gets loaded into the symbol table and mapped program memory. You can combine the options in any order. If no options are specified, all symbolic information and program data is loaded from the file. PL/M-51 object files must contain line number information even if the NOLINES option is specified in the LOAD command.

If the object file is not a valid MCS-51 object file, the ICE-5100 emulator assumes that the file is an ASCII hexadecimal object file and attempts to load the file as such. Symbolic information in an ASCII hexadecimal file is ignored during the LOAD process, and the load options have no effect.

NOTE

The LOAD command only loads into ICE program memory, therefore, you must map sufficient memory to ICE (using the MAP command) before loading your program.

Use the LOAD command to restore the contents of program memory previously saved with the SAVE command.

Examples

1. Load a program from a Series III file.

```
hlt>LOAD :f1:messg
hlt>
```

2. Load a program from a PC disk drive with code only.

```
hlt>LOAD a:messg NOSYMBOLS
hlt>
```

3. Load program symbols and line numbers only.

```
hlt>LOAD a:messg NOCODE
hlt>
```

Cross-Reference

MAP
Pathname
SAVE

LSTEP

Single-steps through user programs by line numbers

Syntax

LSTEP [*number*] [FROM *addr-spec*]

Where:

LSTEP	executes by numbered high-level language statements.
number	is an unsigned integer expression in the current base specifying the number of steps to take. The default increment is 1.
FROM	specifies a starting address where LSTEP is to begin. The default starting address is the current execution point ($).
addr-spec	is a memory location in CODE.

Discussion

The LSTEP (line step) command single-steps through user programs by numbered high-level language statements. The LSTEP command executes the specified number of consecutive statements and halts. A break message displays the current high-level statement number representing the current execution point ($).

The LSTEP command single-steps until it reaches a high-level line number. The LSTEP command does not have to begin at a high-level line number, but the current execution point must be within the NAMESCOPE of a program module.

All instructions executed with the LSTEP command are entered into the trace buffer regardless of the last trace option. Changing the address of the program counter and performing an LSTEP clears the trace buffer of old trace data before new trace data is collected.

NOTE

For large programs, the performance of the LSTEP command can be optimized by specifying a larger symbol table buffer size when invoking the ICE-5100 emulator software.

CLIPS data is collected during an LSTEP command (see Appendix C for information on using the CLIPS assembly).

Examples

1. Step through line number 5.

```
hlt>LSTEP FROM :main_display#5
:MAIN_DISPLAY #6
hlt>
```

2. Step through a range of program lines.

```
hlt>LSTEP 10 FROM :main_display#5
:MAIN_DISPLAY #7
hlt>
```

Cross-Reference

Addr-spec
Expression
ICE*nnn*
ISTEP
Mspace
PRINT

MAP

Displays or sets memory map

Syntax

$$\text{MAP} \left[\textit{[addr-spec]} \begin{Bmatrix} \text{ICE} \\ \text{USER} \end{Bmatrix} \right]$$

Where:

MAP	displays the current memory map.
addr-spec	is a memory location or range of locations.
ICE	directs program memory accesses to ICE-5100 (64KB maximum) emulator memory.
USER	directs program memory accesses to your target hardware. The default is for all program memory, (excluding on-chip ROM when the EA pin is not asserted) to be mapped to USER.

Discussion

The ICE-5100 emulator uses a memory map to direct processor address space to physical memory locations and to control access to mapped program memory. Use ICE memory when there is no target system (USER) or when you want to be able to modify program memory. When mapping to USER, no check is made to ensure that the amount of memory installed in the target system matches the map. It is not possible to perform write operations on program memory that is mapped to USER.

Restrictions

None of the ICE-5100 emulator's data memory is mappable.

Mapping of the on-chip ROM address space is dependent on the state of the EA pin in the target system. All on-chip ROM address space is always mapped to ICE if the EA pin is not asserted or if the ICE-5100 emulator is being used in the stand-alone mode of operation with the CPA.

Specifying the Number of Blocks to be Mapped

The memory map is described to the ICE-5100 emulator in terms of program memory blocks using the address option. Each block is 4KB (4096 bytes). When the starting address does not begin on a block boundary or the last location in the range does not fill a whole block, the ICE-5100 emulator automatically expands the map to the next boundary and reports the expansion.

Examples

1. Display the current memory map. (Example assumes the 252 probe.)

```
hlt>MAP
MAP 0K LENGTH 8K ICE
MAP 8K LENGTH 56K USER
[USER EA PIN = 1]
hlt>
```

2. Map all memory to ICE.

```
hlt>MAP ICE
hlt>
```

3. Map a range of addresses to USER.

```
hlt>MAP 0 LENGTH 8K USER
hlt>
```

4. The following example shows how the ICE-5100 emulator adjusts partitions to match block boundaries.

```
hlt>BASE=10T
hlt>MAP
MAP 0K LENGTH 8K ICE
MAP 8K LENGTH 64K USER
[USER EA PIN = 1]
hlt>MAP 8K TO 10K ICE        /*MAP partition not on a 4 KB boundary*/
WARNING: Map address boundaries changed to match hardware
hlt>MAP
MAP 0K LENGTH 12K ICE
MAP 12K LENGTH 52K USER
[USER EA PIN = 1]
hlt>
```

Cross-Reference

Name
Addr-spec
RESET

MENU

Enables and disables the syntax menu display

Syntax

$$\text{MENU} = \begin{bmatrix} \text{TRUE} \\ \text{FALSE} \\ \textit{boolean-expression} \end{bmatrix}$$

Where:

TRUE	enables the menu display. The initial value is TRUE.
FALSE	disables the menu display.
boolean-expression	evaluates to TRUE (LSB = 1) or FALSE (LSB = 0).

NOTE

<CNTL>V is a toggle switch that changes the setting of MENU.

Discussion

The ICE-5100 emulator menu is a syntax directory on the bottom of the screen. The syntax directory aids in constructing syntactically correct commands. You can, however construct a syntactically correct command that is semantically incorrect, i.e., BASE = 8T. The MENU pseudo-variable lets you enable or disable the menu display.

When MENU = TRUE, the menu displays all of the legal tokens which you can enter at the current cursor position. As you advance the cursor to the next token (with a space or other delimiter), the menu is updated to show legal tokens at the new position. If all the available tokens do not fit on one line, press the TAB key to display more choices. If you enter a token not on the current token list, the ICE-5100 emulator returns a syntax error.

Examples

1. The following example shows the first menu display after entering the ICE-5100 software invocation command. (Example is for the ICE-5100/252 user probe.)

```
---- more ---- Use [TAB] to cycle through prompts.
APPEND ASM BASE BRKREG CAUSE CLEAREOL CLEAREOS COUNT CPU
```

2. Disable the menu display in either of two ways, and re-enable with a third method.

```
hlt> MENU = FALSE
```

or:

```
hlt> MENU = 0
```

or:

```
hlt> <CNTL><V>
hlt>
```

3. The menu changes after entering GO <*space*> to reflect the available options.

```
hlt> GO <space>
FROM ARM FOREVER TIL USING TRACE ; <execute>
```

Mspace

Specifies an MCS-51 memory address space

Discussion

The MCS-51 family of microcontrollers has four physically distinct memory spaces. Specifying a physical memory address by itself is not enough information to access a memory location. Therefore, commands that access memory must specify an *mspace* prefix to declare the area of memory in which the address partition resides. These are accessed by *mspace* keywords as follows:

CODE	is the 64-KB program memory space.
IDATA	is the internal data memory space. This is the default *mspace* if none is specified. This *mspace* overlaps the space accessed by the BIT *mtype*.
RDATA	is 128-byte special function register (SFR) area located between addresses 80H - FFH. Not all addresses are used. The BIT *mtype* can be used to access bit-addressable registers in this *mspace*.
XDATA	is the 64-KB external data memory space.

Use an *mspace* prefix when memory is specified with an address partition. An *mspace* prefix is not needed with symbolic memory accesses because the symbol contains an implicit *mspace*. An *mspace* prefix is also not allowed when accessing a bit location.

Examples

1. The following example writes then reads a byte in IDATA.

```
hlt>BYTE 23 = 41H
hlt>BYTE 23H
IDATA   0023H   41
hlt>
```

2. Modify and display a portion of the external data memory space.

```
hlt>BYTE XDATA 55 LENGTH 5 = 'HELLO'
hlt>BYTE XDATA 55 LENGTH 5
XDATA   0055H   48 45 4C 4C 4F        'HELLO'
hlt>
```

3. Display and modify a bit addressable memory location.

```
hlt> flag1
1
hlt> BIT 78 to 7F = flag1
hlt> BIT 78 LENGTH 8
BIT   0078H  1 1 1 1 1 1 1 1
hlt>
```

Cross-Reference

Mtype
Symbolic Reference

Mtype

Memory types

Discussion

The ICE-5100 emulator requires a memory type (*mtype*) to be specified when accessing memory. The following sections describe the internal and external representations for each *mtype* and the rules for type conversion between *mtypes*.

Use and Display of Mtypes

ADDRESS	is a 16-bit unsigned value, displayed as a WORD. The ICE-5100 emulator does not distinguish between the WORD and ADDRESS *mtypes*. ADDRESS is included to provide compatibility with high-level languages that support this memory type.
BIT	is a one-bit binary value, displayed as 0 or 1.
BOOLEAN	is an eight-bit unsigned value, displayed as FALSE if low order bit is 0 and TRUE if low order bit is 1.
BYTE	is an eight-bit unsigned value, displayed in the current number base.
CHAR	is an eight-bit ASCII value, displayed as a quoted string. A non-printing character is indicated by a period (.).
WORD	is a 16-bit unsigned value, displayed in the current number base.

Type Conversions

Type conversion is necessary when *mtypes* are combined within an expression or a value of one *mtype* is assigned to a variable of another *mtype*.

Type conversion includes length adjustment. This is necessary in two cases:

- A type converted value is stored in an *mtype* shorter than the original: The excess bits are truncated, retaining the low-order ("rightmost") values.
- A type converted value is stored in an *mtype* longer than the original: The excess bits are zero-padded and the value placed in the low-order ("rightmost") bits.

Table 3-15 shows the valid conversions within an expression. An error results from any combination not shown in the table. Table 3-16 shows the valid and invalid conversions by assignment, including the internal conversion path where applicable.

Table 3-15 Type Conversion by Combination as Operands

Operator	Operands	Result
Arithmetic	Unsigned and Unsigned	WORD
(+, −, *, /, MOD)	Unsigned and Character	WORD
	Character and Character	CHARACTER
Logical	Unsigned and Unsigned	WORD
(AND, OR, XOR)	Boolean and Unsigned	BOOLEAN
	Boolean and Boolean	BOOLEAN
	Boolean and Bit	BOOLEAN
	Character and Unsigned	WORD
	Character and Character	CHARACTER
	Bit and Unsigned	WORD
	Bit and Bit	BIT
Relational	Unsigned and Unsigned	WORD
(= =, > =, < =	Boolean and Unsigned	BOOLEAN
>, <, < >)	Boolean and Boolean	BOOLEAN
	Boolean and Bit	BOOLEAN
	Character and Unsigned	WORD
	Character and Character	CHARACTER
	Bit and Unsigned	WORD
	Bit and Bit	BIT
Unary	Unsigned	WORD
NOT, +, −	BIT	BIT
NOT	BOOLEAN	BOOLEAN

Table 3-16 Assignment Type Conversions*

FROM TYPE T1	TO TYPE T2			
	Unsigned	Boolean	Character	Bit
Unsigned (BYTE, WORD, ADDRESS)	Convert to WORD, truncate to T2	IF LSB(T1) = 1 then T2 = TRUE else T2 = FALSE	Convert to WORD, truncate to T2	IF LSB(T1) = 1 then T2 = TRUE else T2 = FALSE
Boolean (BOOLEAN)	INVALID	No conversion necessary	INVALID	IF LSB(T1) = 1 then T2 = TRUE else T2 = FALSE
Character (CHAR)	Convert to WORD, truncate to T2	INVALID	No conversion necessary	INVALID
Bit (BIT)	Convert to WORD, truncate to T2	IF T1 = 1 then T2 = TRUE else T2 = FALSE	INVALID	No conversion necessary

*The value of type T1 is assigned to a variable of type T2 (i.e., *mtype* T2 = *mtype* T1).

Examples

1. This example shows the use of *mtype* keywords to define debug variables in IDATA, and type conversion by assignment. The ASCII code 7150 represents the character string ‘qP’.

```
hlt> DEFINE ADDRESS qry = 7150
hlt> DEFINE CHAR ans = qry
hlt> ans
P
hlt>
hlt> qry = 5071
hlt> ans = qry
hlt> ans
q
hlt>
hlt> qry = ans
hlt> qry
0071
hlt>
hlt> DEFINE BYTE short = ans
hlt> short
71
hlt>
hlt> DEFINE BOOLEAN flag = short
hlt> flag
TRUE
hlt>
hlt> BASE = BINARY
hlt> short
01110001
hlt>
hlt> DEFINE BIT switch = short
hlt> switch
1
hlt>
hlt> short = switch
hlt> short
00000001
hlt>
```

2. This example shows the use of BIT to access locations in IDATA.

```
hlt>BIT 5
BIT      0005H    1
hlt>

hlt>BYTE IDATA 20 = 0
hlt>BIT 0 LENGTH 8
BIT      0000H    0 0 0 0 0 0 0 0
hlt>
```

3. The following example shows the display associated with each mtype for the memory location CODE 0068H, in this case the current execution pointer. The number base at the start of this example is hexadecimal.

```
hlt>ASM $
ADDR  CODE        INSTRUCTION
0068H 7150        ACALL 0350H
hlt>

hlt>ADDRESS CODE $
CODE     0068H    7150
hlt>

hlt>BOOLEAN CODE $
CODE     0068H    TRUE
hlt>

hlt>BYTE CODE $
CODE     0068H    71
hlt>

hlt>CHAR CODE $
CODE     0068H    'q'
hlt>

hlt>BASE = BINARY
hlt>

hlt>WORD CODE $
CODE     0068H    00111000101010000
hlt>

hlt>BYTE CODE $
CODE     0068H    01110001
hlt>
```

Cross-Reference

ADDRESS
BIT
BOOLEAN
BYTE
CHAR
Mspace
WORD

Name

Rules for creating and using names in commands

Syntax

first-character[*following-character*] . . .

Where:

first-character	is any alphabetic character (a-z or A-Z, uppercase and lowercase are equivalent), an underscore (_), a question mark (?), or an at sign (@).
following-character	is any alphabetic character (a-z), an underscore (_), an at sign (@), a dollar sign ($), a question mark (?), or the digits 0-9. The first 40 characters are significant. The maximum length is 255T characters.

Discussion

Names are either keywords that are predefined by the ICE-5100 emulator, symbols created by you in programs, or symbols created by you in commands while debugging those programs. The ICE-5100 emulator lets you create and change debug object names and manipulate user program symbol names when required. (Refer to the DEFINE command in this chapter.)

The ICE-5100 emulator uses translator-generated names as symbols. The names of variables, labels, procedures, and modules are all symbols. When a name is referenced in a command, the ICE-5100 emulator determines whether it is a keyword, a debug object, or a program symbol. Keywords have the highest precedence, debug objects are next, and program symbols are searched last.

Keywords are predefined and cannot be changed or removed. The emulator does not permit a debug object to have the same name as a keyword.

Program symbols are loaded with the object file. A program symbol may duplicate a keyword or debug object name. During a debugging session, precede a program variable that duplicates a command with a double-quote operator (") to force the emulator to look in the program symbol table for a reference. A symbolic reference must also include the double-quote operator when the program symbol name duplicates a debug symbol name. (Refer to the Symbolic references entry in this chapter for more information on the quote operator.)

Name (continued)

NOTE

$ is not a significant symbol. Firstcharacter is the same as first$character.

_, @ and ? are significant symbols. PROC_TWO is different from PROCTWO.

Cross-Reference

DEFINE
Keywords
Symbolic references

NAMESCOPE

Displays or sets the current symbol reference point

Syntax

NAMESCOPE [= *addr-spec*]

Where:

NAMESCOPE	displays the reference address (pointer) that determines the set of visible program objects.
addr-spec	is a numeric or symbolic reference to a program location. The *addr-spec* evaluates to type WORD.

Discussion

The NAMESCOPE pseudo-variable is a pointer to a location in your program. When you load a program, NAMESCOPE is set to the address of the prologue of the main module. NAMESCOPE changes in three cases:

- When it is set with the NAMESCOPE pseudo-variable.
- When a program is reloaded, NAMESCOPE is reset to the prologue of the main module.
- Whenever the execution point changes (HALT, $, ISTEP, LSTEP, or execution break) while DYNASCOPE = TRUE, NAMESCOPE is changed to the execution point.

The ICE-5100 emulator uses the NAMESCOPE address as a reference point to determine the amount of qualification required to identify an object in the program. A fully-qualified reference to a symbol includes the module name and the names of all procedures that enclose the symbol in order from outer-most to inner-most (:module.procedure.variable). A fully-qualified reference is always valid.

A partially-qualified reference omits the module name and one or more of the outer procedure names. The emulator looks up a partially qualified reference as follows:

1. If the NAMESCOPE address is the beginning address of a procedure, the symbols defined in that procedure are checked, and then the symbols in each enclosing procedure are checked.
2. If the symbol is not found, each enclosing procedure, main module, and all public symbols are searched (in that order) until the symbol is found or until the search fails.

NAMESCOPE (continued)

Restrictions

Due to a limitation of the debug information provided in 8051 absolute object files, NAMESCOPE does not recognize procedure blocks. Therefore, in order for NAMESCOPE to work properly, NAMESCOPE must be set to the beginning of the procedure block. For example, setting NAMESCOPE = :MAIN.PROC1 would allow partially-qualified references to symbols in PROC1.

NOTE

Loading user code with the NOCODE or NOSYMBOLS option disallows use of the NAMESCOPE feature.

Examples

The following examples assume that the user program has two modules with enclosed procedures and variables, structured as follows:

```
adder (MODULE)
    operand_count (PROCEDURE)
        number_of_ops (BYTE VARIABLE)
    error_check (PROCEDURE)
        error_number (BYTE VARIABLE)
display_character (MODULE)
    char_check (PROCEDURE)
        char_count (BYTE VARIABLE)
    output_char (PROCEDURE)
```

1. Access the variable NUMBER OF OPS from the module ADDER.

```
hlt> operand_count.number_of_ops
3
hlt>
```

2. Access the variable CHAR_COUNT in a different module. You need a fully qualified reference that includes the module name and all enclosing procedure names.

```
hlt> :display_character.char_check.char_count
26
hlt>
```

3. To allow a short (partially-qualified) reference to CHAR_COUNT, change NAMESCOPE as shown.

```
hlt> NAMESCOPE = :display_character.char_check
hlt> char_count
26
hlt>
```

Cross-Reference

$
Addr-spec
DYNASCOPE
Pseudo-variable

NUMTOSTR

A function that converts an expression into ASCII code

Syntax

NUMTOSTR (*expression*)

Where:

NUMTOSTR	converts *expression* into its ASCII representation.
expression	is any valid combination of values and operations.

Discussion

The NUMTOSTR function is useful with other string functions in debug procedures. The conversion is displayed on the screen but does not alter memory. The current number base is used for conversion.

Example

1. Display the variables as an ASCII string.

```
hlt> BASE=10T
hlt> DEFINE CHAR temp_s = NUMTOSTR(3 * 4 + 3)
hlt> temp_s
15
hlt> CONCAT('Answer is',temp_s)
Answer is 15
```

Cross-Reference

Expression

Paging

Controls screen display

Syntax

$$\left\{\begin{matrix}\text{F}\\\text{P}\\\text{L}\end{matrix}\right\}$$

Where:

F	(fast) writes data continuously to the screen.
L	(line) writes a single line to the screen at a time.
P	(page) writes to the screen one screen full at a time; initial condition.

Discussion

Some ICE-5100 emulator commands display more information than will fit on a single screen. The screen display controls limit the amount of information displayed to the screen at one time. Enter the paging commands while the information is scrolling.

In the PAGE and LINE modes, a message is displayed at the bottom of the screen to inform you of addditional information waiting to be displayed. The format of the message is

```
Enter L[ine], F[ast], or P[age] to change mode, any other key to continue
```

The prompt (hlt> or emu>) appears when the final screenful of information has been displayed.

The screen display controls are effective only during the display process.

Cross-Reference

Control keys

Pathname

Specifies the name and the location of a file

Syntax

Series III Hosts

[:*device*:]*filename*

Series IV Hosts

$$\left\{ \begin{array}{l} \text{[/}\textit{directory}\text{] [. . .]/}\textit{filename} \\ \text{:}\textit{device}\text{:[}\textit{filename}\text{]} \end{array} \right\}$$

Where:

device	is a device such as a disk drive or line printer. Values for *device* are: F*n* — Disk drive number (0 < *n* < 9T) LP — Line printer (no *filename* necessary)
filename	is the name of the file. The filename can contain up to six alphanumeric characters, plus a three-digit extension.
directory	is the name of a directory.

IBM PC Hosts

[*device*:] [/*directory*] [. . .] /*filename*

Where:

device	is a hard disk or floppy disk drive. Values for device are A, B, C, D, etc.
directory	is the name of a directory.
filename	is the name of a file. The filename can contain up to eight alphanumeric characters, plus up to a three-number or three-letter extension (e.g., myprog.plm).

PRINT

Formats and displays the contents of the trace buffer

Syntax

```
                             ┌ ┌ CLEAR               ┐ ┐
                             │ │ ALL                 │ │
        ┌ ┌ CLIPS   ┐ ┐      │ │ OLDEST [expression] │ │
PRINT   │ │         │ │      │ │ NEWEST [expression] │ │
        └ └ NOCLIPS ┘ ┘      │ │ NEXT [expression]   │ │
                             │ │ LAST [expression]   │ │
                             └ └ addr-spec           ┘ ┘
```

Where:

PRINT	prints a single trace frame starting with the oldest frame. This is the default display condition.
CLIPS	specifies that all subsequent PRINT commands will display the user trace clips status. CLIPS remains in effect until NOCLIPS is specified.
NOCLIPS	specifies that all subsequent PRINT commands will not display the user trace clips status. NOCLIPS remains in effect unless CLIPS is specified. NOCLIPS is the initial default.
CLEAR	clears the trace buffer of all data.
ALL	specifies that the entire trace buffer contents are to be displayed starting with the oldest frame.
OLDEST	prints the specified number of trace frames starting with the oldest (frame 0) frame in the trace buffer. If no argument is specified, one frame is displayed.
NEWEST	prints the specified number of trace frames ending with the newest frame in the trace buffer. If no argument is specified, one frame is displayed.
NEXT	prints the specified number of trace frames starting with the current frame (the last displayed frame) in the trace buffer.
LAST	prints the specified number of trace frames ending at the current frame (the last displayed frame) in the trace buffer.
expression	is a number between 1 and +253T or a variable that converts to a numeric value between 1 and +253T.
addr-spec	is a single instruction or range of instruction frames between 0 and +253T.

PRINT (continued)

Discussion

Use the PRINT command to display a disassembled listing of information in the trace buffer. Modify the display using the listed syntax options. Trace information is collected during emulation (GO, LSTEP, ISTEP) and placed into a trace buffer that is 254 frames long (0 to 253T).

The trace buffer has three pointers:

- *oldest* — points to the entry or frame which is chronologically oldest
- *newest* — points to the entry or frame which is chronologically newest
- *current* — points to the entry or frame which was the last one to be displayed

If trace data has been collected since the last PRINT command, *current* = *oldest* (circular trace buffer), and more than 254 frames of trace data were collected, then the most recent 254 trace frames will be contained between *oldest* and *newest* in the trace buffer. Trace frames which are outside of the range *oldest* and *newest* (empty frames) will not be displayed.

The format of the display has the following headings:

FRAME	represents the frame number.
ADDRESS	is the memory location of the opcode.
CODE	is the hexadecimal opcode value.
INSTRUCTION	is the mnemonic corresponding to the opcode, followed by its operands in hexadecimal format.
CLIPS	is optional, and displays clips status. Refer to Appendix C for information on using the clips assembly.

Use the EDIT TRACE command to edit the trace data (refer to the EDIT entry in this chapter).

Examples

1. Print the newest instruction in the trace buffer.

```
hlt>PRINT NEWEST
FRAME     ADDR  CODE      INSTRUCTION
(34)      0003H 22        RET
hlt>
```

2. Print a range of instructions.

```
hlt>PRINT 1 to 5
FRAME     ADDR  CODE          INSTRUCTION
(01)      0027H 7401          MOV    A,#001H    ; +001T
(02)      0029H 26            ADD    A,@R0
(03)      002AH F6            MOV    @R0,A
(04)      002BH 50DD          JNC    0000AH     ; $ - 21H
(05)      000AH E50B          MOV    A,I
hlt>
```

3. Print the oldest two instructions in the trace buffer showing the clips status.

```
hlt>PRINT CLIPS OLDEST 2
FRAME     ADDR  CODE          INSTRUCTION                       CLIPS (76543210)
(00)      021AH 7509C7        MOV  TEXT_PTR + 1T, #0C7H               10101111
(01)      021DH 750A02        MOV  TEXT_CNT, #002H                    00100010
hlt>
```

Cross-Reference

EDIT
GO
ISTEP
LSTEP
TRCREG

PROC

Is a debug procedure

Syntax (two forms)

1. Define a PROC:

```
DEFINE PROC name = DO
                ICE-5100 emulator command(s)
            END
```

2. Display a PROC:

```
PROC name
```

Where:

DEFINE	defines or redefines a debug object.
PROC	is the keyword for creating or displaying a debug procedure.
DO	begins a block construct. DO must be the first statement in a PROC definition.
name	is the user-defined name of the procedure.
ICE-5100 emulator command(s)	are one or more ICE-5100 emulator commands except INCLUDE, EDIT, and HELP.
END	terminates the block construct.

Discussion

Use debug procedures to block several commands into a single function. The only limit on the size of procedures is the amount of memory space available. Although a debug procedure is not executed until it is invoked, the ICE-5100 emulator checks the syntax when the procedure is defined. Debug procedures can be defined or modified with the editor.

Debug procedures can be defined within other debug procedures. The inner debug procedure is not visible to the ICE-5100 emulator until the outer debug procedure is executed. Once debug procedures become visible to the ICE-5100 emulator, they are always global, even when nested inside other debug procedures.

NOTE

You must define a debug object before it can be referenced by a debug procedure.

Save debug procedures for future use with the PUT or APPEND command. Recall saved procedures with the INCLUDE command. Delete debug procedures with the REMOVE command. Display a list of currently defined debug procedures with the DIR command.

Use a debug procedure during emulation by calling the PROC from a break register (BRKREG). Use the RETURN command to return a debug procedure value. An error occurs when a debug procedure is called and does not return a value as expected. The syntax of the RETURN command is as follows:

RETURN [*expression*]

The *expression* must be a Boolean or an expression that evaluates to a Boolean.

Referencing Parameters in Debug Procedures

Use the percent sign (%) to tell the ICE-5100 emulator that you will furnish parameters when you invoke the debug procedure.

%NP	A predefined system parameter equal to the number of parameters passed in the debug procedure.
%*number*	A parameter *number* which selects that parameter from the list following the debug procedure invocation. Numbers range consecutively from 0 through 99.
%(*expression*)	Used instead of *number* but requires parentheses. Must evaluate to a number between 0 and 99.

You cannot omit defined parameters. For example, if the debug procedure is defined as having three parameters, passing the procedure only two parameters, such as (I, ,J) or (I,J), is illegal.

Examples

1. Define and execute a simple debug procedure that averages three parameters.

```
hlt> DEFINE PROC average = DO ((%0 + %1 + %2)/%NP) END
hlt> average (10T,2,3)
5
hlt>
```

2. Define a more complex procedure.

```
hlt> DEFINE PROC show_array = DO
.hlt> DEFINE BYTE leng = %0                    /* second parameter is number to print */
.hlt> DEFINE BYTE i                                              /* array index */
.hlt> i = 0
.hlt> WRITE 'Partial dump of byte_array:'
.hlt> COUNT %0
..hlt> WRITE USING('H,2," ",>')BYTE .:main_prog.byte_array + i
..hlt>                                  /*byte_array is a user-program array */
..hlt> i = i + 1
..hlt> ENDCOUNT
.hlt> WRITE 'Done!'
.hlt> END                                               /* of proc show_array */
hlt>
```

3. Display a debug procedure.

```
hlt> proc show_array
define proc show_array = do
define byte leng = %0
define byte i
i = 0
write 'Partial dump of byte_array:'
count %0
write using('H,2," ",>')byte .:main_prog.byte_array + i
i = i + 1
endcount
write 'Done!'
end
hlt>
```

4. Execute a debug procedure.

```
hlt> show_array(5)
Partial dump of byte_array:
A5 6B 02 0F 1E
Done!
hlt>
```

5. Modify a debug procedure.

```
hlt> EDIT average
hlt>
```

6. Delete a debug procedure.

```
hlt> REMOVE average
hlt>
```

Cross-Reference

APPEND
BRKREG
DEFINE
EDIT
Name
PUT
REMOVE

Pseudo-variable

A system-defined variable

Pseudo-variables are a cross between commands and variables. Pseudo-variables initiate operations. Like variables, pseudo-variables are named, have a value, can be assigned and displayed, and can be used in expressions, as shown in the following command line:

```
hlt>IF BASE == 16T THEN BASE = 10T
```

Pseudo-variables are predefined by the ICE-5100 emulator and cannot be removed.

The ICE-5100 emulator's pseudo-variables are the following:

$
BASE
CURX
CURY
DYNASCOPE
ERROR
NAMESCOPE
RBANK
SYMBOLIC
SYNCSTART
TEMPCHECK
VERIFY

For information on a specific pseudo-variable, refer to its entry in this chapter.

PUT

Saves debug object definitions to a file

Syntax

```
                  ⎧ DEBUG                          ⎫
                  ⎪     ⎧ BRKREG    ⎫              ⎪
                  ⎪     ⎪ TRCREG    ⎪              ⎪
PUT pathname      ⎨     ⎨ PROC      ⎬  [, . . .]   ⎬
                  ⎪     ⎪ LITERALLY ⎪              ⎪
                  ⎪     ⎪ mtype     ⎪              ⎪
                  ⎩     ⎩ name      ⎭              ⎭
```

Where:

PUT	opens the file specified in *pathname* to store debug objects.
pathname	is the fully-qualified reference to the file into which you want to save debug objects.
DEBUG	appends all currently defined debug definitions to the file named in *pathname*.
BRKREG . . . LITERALLY	limits objects appended to a file to the type specified.
mtype	is ADDRESS, BIT, BOOLEAN, BYTE, CHAR, or WORD.
name	is the user-defined name of a single debug object to be saved.

Discussion

Use the PUT command to create a file and save debug definitions to it. If the named file already exists, you are prompted with the message "Overwrite existing file? (y or [n])." If the response is anything other than y, the PUT command is aborted; otherwise, PUT overwrites the named file. (Use the APPEND command to add to a file rather than overwriting it.) The values of debug memory types are not saved. For use in future debugging sessions, use the INCLUDE command to retrieve the contents of the file. Only the most recent definition is restored by the INCLUDE command.

PUT (continued)

Examples

1. DEFINE debug objects, PUT them in a new file, retrieve them from file.

```
hlt> DEFINE TRCREG trace1 = #12
hlt> DEFINE BRKREG brk1 = #23
hlt> PUT dvars.01 trace1, brk1
hlt> INCLUDE dvars.01
DEFINE TRCREG trace1 #12
DEFINE BRKREG brk1 = #23
hlt>
```

2. ReDEFINE a debug object, PUT in the same file as the old, use the file.

```
hlt> DEFINE BRKREG brk1 = #17
hlt> PUT dvars.01 brk1
OVERWRITE EXISTING FILE? (Y OR [N])
hlt> y
hlt> INCLUDE dvars.01
DEFINE BRKREG brk1 = #17
hlt>
```

Cross-Reference

APPEND
BRKREG
INCLUDE
LITERALLY
Mtype
PROC
REMOVE
TRCREG

R0 - R7

Displays or modifies general purpose registers

Syntax

$$\begin{Bmatrix} \text{R0} \\ \text{R1} \\ \text{R2} \\ \text{R3} \\ \text{R4} \\ \text{R5} \\ \text{R6} \\ \text{R7} \end{Bmatrix} [= \textit{expression}]$$

Where:

R0-R7	displays the current value of the specified general purpose register.
expression	resolves to an 8-bit value.

Discussion

The MCS-51 family of mircontrollers contain four banks of eight general purpose registers numbered R0 through R7. Depending on the bank selected (via the RBANK command or by writing bits to RS1 and RS0 in the PSW register), the R0 - R7 command displays or modifies the contents of the specified register.

Example

1. Display the contents of R1.

```
hlt> R1
02H
hlt>
```

2. Change the contents of R1.

```
hlt> R1 = 05H
hlt>
```

Cross-Reference

RBANK
REGS

RBANK

Displays or modifies the
Register Bank Select

Syntax

RBANK [= *expression*]

Where:

RBANK	displays the contents of the Register Bank Select (flags RS0 and RS1 in the Program Status Word).
expression	can be a valid number, or a numeric expression that yields a valid number between 0 and 3T to change the contents of the Register Bank Select (flags RS0 and RS1 in the Program Status Word).

Discussion

The RBANK command without an expression displays the contents of the Register Bank Select. *expression* is a new value for Register Bank Select. An MCS-51 family microcontroller contains four banks of eight general purpose registers. RBANK selects one of these as the currently active bank.

Example

1. Display the contents of the RBANK.

```
hlt> RBANK
3
hlt>
```

2. Change the RBANK to bank 2.

```
hlt> RBANK = 2
hlt>
```

Cross-Reference

Pseudo-variable
REGS
R0-R7

REGS

Displays registers and flags

Syntax

```
REGS
```

Discussion

The REGS command displays the contents of the general purpose registers, special function registers, and program status word flags. Table 3-17 lists the names of the registers displayed by the REGS command.

For more information on accessing registers, see the SFR section in your user probe supplement.

Table 3-17 Registers Displayed by the REGS Command

Symbol	Name
$	Program counter
ACC	Accumulator
SP	Stack pointer
DPTR	Data pointer
TM0	Timer 0
TM1	Timer 1
TM2	Timer 2 (processor dependent)
RBANK	Register bank select
R0-R7	Registers 0 through 7
CY	Carry flag
AC	Auxiliary carry flag
F0	Flag 0
RS1	Register bank select bit 1
RS0	Register bank select bit 0
OV	Overflow flag
P	Parity flag

REGS (continued)

Example

1. Display selected registers and flags (example lists the registers and flags of the 80C252 microcontroller).

```
hlt> REGS
$     =0026H   TM0     =0000H   R0  =  02H   R4  =  0CH
ACC  =  0AH   TM1     =0000H   R1  =  FEH   R5  =  00H
SP   =  07H   TM2     =0050H   R2  =  06H   R6  =  03H
DPTR =10C6H   RBANK  =   00H   R3  =  B0H   R7  =  CAH
FLAGS: CY=1   AC=0    F0=0    RS1=0   RS0=0   OV=0   P=0
hlt>
```

Cross-Reference

$
DPTR
RBANK
R0-R7
TM0 - TM2

REMOVE

Removes user program symbols
or debug objects

Syntax

```
           ┌ DEBUG                                  ┐
           │ SYMBOLS                                │
           │   ┌            ┐ ┌       ┐             │
           │   │ BRKREG     │ │       │             │
           │   │ TRCREG     │ │       │             │
REMOVE    <    │ PROC       │ │       │             >
           │   │ LITERALLY  │ │ ,...  │             │
           │   │ mtype      │ │       │             │
           │   │ name       │ │       │             │
           │   └            ┘ └       ┘             │
           └                                        ┘
```

Where:

REMOVE	is the keyword used to remove the specified debug objects or user program symbols.
DEBUG	deletes all currently defined debug definitions.
SYMBOLS	deletes all currently defined user program symbols.
BRKREG . . . LITERALLY	limits objects deleted to the type specified.
mtype	is ADDRESS, BIT, BOOLEAN, BYTE, CHAR, or WORD.
name	is the user-defined name of a single debug object or program symbol to be deleted.

Discussion

Use the REMOVE command to delete user program symbols and debug object definitions individually or as a class. Deleting a BRKREG or TRCREG which was used in the last GO command does not deactivate those break or trace points in the ICE-5100 emulator hardware.

Examples

1. Remove a single BRKREG definition named firstbrk and all debug procedure definitions.

```
hlt>REMOVE firstbrk, PROC
hlt>
```

REMOVE (continued)

Cross-Reference

BRKREG
DEFINE
INCLUDE
LITERALLY
Mtype
PROC
TRCREG

REPEAT

Groups and executes commands forever or until an exit condition is met

Syntax

```
REPEAT
    ⎧ ICE-5100 emulator command(s) ⎫
    ⎨ WHILE boolean-condition      ⎬
    ⎩ UNTIL boolean-condition      ⎭
END[REPEAT]
```

Where:

REPEAT	is a loop control keyword.
ICE-5100 emulator command(s)	is any of the ICE-5100 emulator commands except EDIT, HELP, and INCLUDE.
WHILE *boolean-condition*	continues to execute while *boolean-condition* is TRUE. Execution halts when the WHILE condition is FALSE.
UNTIL *boolean-condition*	halts execution when the *boolean-condition* is TRUE.
END[REPEAT]	ends the REPEAT block and starts execution. The optional REPEAT keyword is used to label the block type.

Discussion

REPEAT blocks are executed immediately after you enter the END statement. REPEAT blocks not containing WHILE or UNTIL clauses are executed forever or until aborted with <CNTL>C (<CNTL><BREAK> on IBM PC hosts). REPEAT blocks containing WHILE or UNTIL exit when the test condition is satisfied.

REPEAT (continued)

Example

1. The following example repeats a command sequence until the program variable is greater than 100.

```
hlt>var1 = 1
hlt>REPEAT
.hlt>var1 = var1 * 2
.hlt>WRITE 'var1 =',var1
.hlt>UNTIL var1 > 100
.hlt>ENDREPEAT
var1 = 2
var1 = 4
var1 = 8
var1 = 16
var1 = 32
var1 = 64
var1 = 128
hlt>
```

RESET

Resets ICE-5100 emulator functions

Syntax

```
RESET { ICE [BAUD(baud-rate) ] [CHANNEL(number) ] }
      { CHIP                                      }
```

Where:

ICE	resets the controller pod to its initial state and reloads the ICE-5100 emulator software. If either the BAUD or CHANNEL option is not specified, then the controller is reset using the previous baud rate or channel. A RESET CHIP is performed and the memory map is reset to its previous state.
BAUD	resets the serial communications baud rate to the ICE-5100 controller pod.
(baud-rate)	specifies the serial communications baud rate. Valid baud rates are 300, 1200, 9600, and 19200.
CHANNEL	reinitializes the serial channel to which the ICE-5100 emulator is connected.
(number)	specifies the serial channel of the host that the ICE-5100 emulator is connected. *number* can be 1 (default channel) or 2.
CHIP	resets the emulation processor. The interrupt-in-progress flag is reset with this command. The stack pointer is reset to its default setting (07H), ports P0, P1, P2, and P3 are reset to 0FFH, and all other on-chip registers are reset to 00H. On-chip data memory is not affected.

Discussion

The RESET command resets functions of the ICE-5100 emulator. In general, a RESET ICE is needed following a probe interface error or whenever serial communication is not properly established between the host and probe. A RESET CHIP can be used to return the emulation processor to its initial power-up state.

Examples

1. Reset the controller pod of the ICE-5100 emulator and reinitialize the baud-rate to 19200.

```
hlt>RESET ICE BAUD (19200)
hlt>
```

RESET (continued)

2. Reset all functions of the ICE-5100 emulator using the previous baud rate and channel.

```
hlt>RESET ICE
hlt>
```

3. The following example illustrates the need to reset the ICE-5100 emulator's serial channel via the RESET ICE command.

```
hlt>REGS
ERROR # 547
Probe or serial port not responding.
hlt>RESET ICE CHANNEL (2)
hlt>
```

4. Reset the probe to its power-up state.

```
hlt>RESET CHIP
hlt>
```

Cross-Reference

ICE*nnn*
MAP

SAVE

Saves the contents of program memory to a file

Syntax

SAVE *pathname addr-spec*

Where:

pathname	is the fully-qualified reference to the file to which you are saving memory.
addr-spec	is a memory location or range of locations within the ICE-5100 emulator's 64 KB program memory space.

Discussion

The SAVE command enables you to save the contents of program memory in a format that allows it to be loaded back into memory with the LOAD command. This command is useful when you are making assembly-level patches to your program code. The patches may be saved for use in future debug sessions. Symbolic information is not saved with the SAVE command.

If the file specified by *pathname* already exists, you are prompted with the message "Overwrite existing file? (y or [n])." If your response is anything other than y, the SAVE command is aborted; otherwise, the existing file is overwritten.

Examples

1. Save a segment of the ICE-5100 emulator's 64 KB program memory space.

```
hlt> SAVE C:\SCRAPDIR\SAVIT 18H TO 0FFH
hlt>
```

2. Save a large portion of the target system's program memory space.

```
hlt> SAVE C:\SCRAPDIR\SAVIT 0H LENGTH 8K
hlt>
```

Cross-Reference

LOAD
MAP

STRLEN

Returns the length
of a string

Syntax

STRLEN (*string-ref*)

Where:

string-ref	is a string reference which can be characters enclosed in apostrophes, a string expression using the CONCAT, NUMTOSTR, or SUBSTR function, or a reference to a type CHAR debug variable.

Examples

1. Return the number of characters in the string “hello”.

```
hlt> STRLEN('hello')
5
hlt>
```

2. Return the number of characters in the debug variable “temp”.

```
hlt> DEFINE CHAR temp = 'hello'
hlt> STRLEN(temp)
5
hlt>
```

Cross-Reference

Expresssion

STRTONUM

String to number conversion function

Syntax

STRTONUM (*string-ref*)

Where:

string-ref	can be numbers enclosed in apostrophes, a string expression using the CONCAT, NUMTOSTR, or SUBSTR function, or a reference to a type CHAR debug variable.

Example

1. Convert a string to a WORD variable.

```
hlt> DEFINE WORD num-var = STRTONUM('23456')
hlt> num-var
23456
hlt>
```

Cross-Reference

Expression

SUBSTR

Returns a portion of a string

Syntax

SUBSTR (*string-ref*, *start*, *length*)

Where:

string-ref	can be characters enclosed in apostrophes, a string expression using the CONCAT, NUMTOSTR, or SUBSTR function, or a reference to a type CHAR debug variable.
start	is an expression with a value from 1 through 254T that specifies the index of the first character in the substring.
length	is an expression with a value from 1 through 254T that specifies the number of characters required by the substring.

Discussion

The SUBSTR function lets you observe portions of a string. The SUBSTR function returns the substring *length* starting at the character indexed by *start*. If the index is out of range, the null string (a blank) is returned.

Examples

1. Return a portion of a string starting at position three.

```
hlt> SUBSTR('abcdef',3,2)
cd
hlt>
```

2. If *start* is valid but *length* is longer than the remaining characters in the string, all of the rest of the string is returned.

```
hlt> SUBSTR('abcdef',3,15)
cdef
hlt>
```

Cross-Reference

Expression

SYMBOLIC

Enables or disables symbolic display

Syntax

```
           ┌     ┌ TRUE               ┐ ┐
SYMBOLIC   │  =  ┤ FALSE              ├ │
           └     └ boolean-expression ┘ ┘
```

Where:

SYMBOLIC	displays the current setting.
TRUE	turns the SYMBOLIC option on. The initial value is TRUE.
FALSE	turns the SYMBOLIC option off.
boolean-expression	evaluates to FALSE (LSB = 0) or TRUE (LSB = 1).

Discussion

Use the SYMBOLIC pseudo-variable to control the symbolic display of disassembled mnemonics, memory, and break and trace addresses. If the user program contains a large number of symbols, setting SYMBOLIC to FALSE increases the speed of the ASM and PRINT commands, by inhibiting symbol table look-up.

Examples

1. Display the current setting.

```
hlt> SYMBOLIC
TRUE
hlt> ASM#5
     ADDR     CODE      INSTRUCTION
(:MAIN_DISPLAY.INIT)
     000EH    755322    MOV BUFF_SIZE,22H;+034T
hlt>
```

2. Eliminate symbol table look-up.

```
hlt> SYMBOLIC=FALSE
hlt> ASM#5
     ADDR     CODE      INSTRUCTION
     000EH    755322    MOV     53H,22H;+0034T
hlt>
```

Cross-Reference

Pseudo-variable

Symbolic references

References to program objects

Syntax (six forms)

1. References to program modules:

   :module-name

2. References to program labels:

   [*:module-name.*] [*procedure-name.*] [. . .]*label-name*

3. References to procedures:

   [*:module-name.*] [*procedure-name.*] [. . .]*procedure-name*

4. References to line numbers:

   [*:module-name*] *#line-number*

5. References to variables:

   [*:module-name.*] [*procedure-name.*] [. . .]*variable-name*

6. Changing the value of a variable:

   variable-name = expression

Where:

label-name *module-name* *procedure-name* *label-name* *variable-name*	are the names of program objects that follow the rules for identifiers.
line-number	is one or more decimal digits.
variable-reference	is a reference to a variable.
expression	converts, if necessary, to the type of the variable in the *variable-reference*.

Discussion

You can access the memory symbolically with user-program symbols and variables. NAMESCOPE determines the amount of qualification necessary for a symbolic reference. Precede the symbolic reference with a dot (.) to display the address of a variable.

The value of a program variable can be modified by symbolically accessing memory and assigning a new value. User-program symbol types can be overridden by specifying the desired *mtype* followed by a dot (.) preceding the symbol. All program objects have associated with them an implicit *mspace*.

Examples

1. Display the address of a program procedure.

```
hlt> :main_display
CODE 000EH
hlt>
```

2. Display, then change the value of the variable temp in the procedure rotate.

```
hlt> rotate.temp
20
hlt> rotate.temp = 5
hlt>
```

3. Find the address of the variable temp.

```
hlt> .rotate.temp
IDATA 0009H
hlt>
```

4. Display the BYTE variable temp as a WORD.

```
hlt> WORD.rotate.temp
IDATA   009H   0005
hlt>
```

Cross-Reference

Expressions
Mspace
Mtype
NAMESCOPE

SYNCSTART

Enables and disables synchronous emulation of multiple Intel emulators

Syntax

```
            ┌     ┌ TRUE               ┐ ┐
SYNCSTART   │  =  ┤ FALSE              ├ │
            │     └ boolean-expression ┘ │
            └                            ┘
```

Where:

SYNCSTART	displays the status of the SYNCSTART pseudo-variable.
TRUE	enables the SYNCSTART pseudo-variable.
FALSE	disables the SYNCSTART pseudo-variable; initial condition.
boolean-expression	evaluates to a FALSE (LSB = 0) or TRUE (LSB = 1).

Discussion

Use the SYNCSTART pseudo-variable to execute multi-ICE start and stop functions with the following Intel emulators:

- ICE-49 emulator
- ICE-51 emulator
- ICE-85 emulator
- ICE-86 emulator
- ICE-88 emulator
- VLSiCE-96 emulator

SYNCSTART controls an external party-line signal called SYNC-0. Once all emulators are connected together (via the multi-ICE connector on each unit), none begin emulation until the last one has been directed to start. The first emulator halting emulation causes all other emulators to halt.

The multi-ICE connector on the emulators output the Intel-standard SYNC-0 signal, and the shell is ground. Connect the SYNC-0 and ground of all units together.

When SYNCSTART is FALSE, the ICE-5100 emula neiter dres or reacts to the signal on SYNC-0. Thus, it is effectively disconnected from the line. The ICE-5100 emulator begins emulation immediately when the GO command is given.

When SYNCSTART is TRUE, the start of emulation is qualified by the level on the SYNC-0 line. When the GO command is executed, the ICE-5100 emulator indicates to the other emulators on the line that it is ready to start emulation. Emulation does not start if the other emulators connected on the line are still halted. If this happens, the ICE-5100 emulator responds with the message:

```
    Waiting for external synchronization
```

When the last emulator issues the GO command, all emulators detect the change on SYNC-0 and begin emulation at the same time. All connected ICE's must have SYNCSTART = TRUE before multi-ICE emulation is enabled.

The HALT command recalls the GO request if emulation did not start due to SYNCSTART and SYNC-0 combination.

The ISTEP and LSTEP commands are disabled when SYNCSTART is TRUE.

Table 3-18 lists the electrical characteristics for the multi-ICE (SYNC-0) connector.

Examples

1. Display the current value for SYNCSTART.

```
hlt> SYNCSTART
FALSE
hlt>
```

Table 3-18 Electrical Characteristics for SYNC-0

Parameter	Values
Input voltage (maximum)	5.5 V
Input threshold voltages	
Logic high (typical value)	2.2 V
Logic low (typical value)	1.3 V
Input current (maximum)	
I_{il}	2.0 ma
I_{ih}	50 μa

SYNCSTART (continued)

2. Set SYNCSTART to TRUE (enable multi-ICE emulation).

```
hlt> SYNCSTART = TRUE
hlt> GO FOREVER
Waiting for external synchronization
emu>
Probe stopped at 0F2AH because of external break.
hlt>
```

Cross-Reference

GO
HALT
Pseudo-variable

TEMPCHECK

Controls user probe temperature checking

Syntax

$$\text{TEMPCHECK} \left[= \left\{ \begin{array}{l} \text{TRUE} \\ \text{FALSE} \\ \textit{boolean-expression} \end{array} \right\} \right]$$

Where:

TEMPCHECK	displays the current status of TEMPCHECK.
TRUE	sets TEMPCHECK to TRUE; initial condition.
FALSE	sets TEMPCHECK to FALSE.
boolean-expression	is an expression that evaluates to either 0 (FALSE) or 1 (TRUE).

Discussion

The TEMPCHECK pseudo-variable controls the user probe temperature checking. When TEMPCHECK is TRUE, the user probe temperature is polled at regular intervals and once at the beginning of each new command line. If the ambient room temperature exceeds approximately 113° Fahrenheit (45° Celsius), a warning message is displayed following each new command.

Setting TEMPCHECK to FALSE disables temperature checking and displaying of the warning message.

NOTE

Setting TEMPCHECK to FALSE will not stop the ICE-5100 emulator from shutting down in cases of extreme heat (136° Farenheit or 58° Celsius).

Examples

1. Display the status of TEMPCHECK.

```
hlt> TEMPCHECK
TRUE
hlt>
```

2. Modify the state of TEMPCHECK.

```
hlt> TEMPCHECK = FALSE
```

or

```
hlt> TEMPCHECK = 0
hlt>
```

Cross-Reference

Pseudo-variable

TM0 — TM2

Displays and modifies the contents of the timer/counter registers

Syntax

$$\left\{\begin{array}{l}\text{TM0}\\\text{TM1}\\\text{TM2}\end{array}\right\}\quad [= \textit{expression}]$$

Where:

TM0 — TM2	are the timer/counter registers.
expression	is a 16-bit value or an *expression* that evaluates to a 16-bit value.

Discussion

Use the TM*x* command without expression to display the contents of the 16-bit timer/counter registers on the microcontroller. (Only the 8052, 8032 and 80C252 have TM2.) Contents are displayed or changed in the current number base. Use *expression* to change the contents of the designated timer/counter register.

Do not insert a space between TM and the number of the register. For example, TM 1 is incorrect.

Example

1. Modify and display the contents of TM0.

```
hlt> TM0 = 100H
hlt> TM0
0100
hlt>
```

Cross-Reference

REGS

TRCREG

Contains trace specifications

Syntax (two forms)

1. Define a TRCREG:

```
                              [DO]
DEFINE TRCREG name =  { [OUTSIDE] [PAGE] address     }
                      { FROM [address] [TIL address] }
                              [END]
```

2. Display a TRCREG:

```
TRCREG name
```

Where:

DEFINE	defines or redefines a debug object.
TRCREG	is the keyword for creating or displaying a trace register.
name	is the name of a debug trace register.
OUTSIDE	causes the ICE-5100 emulator to recognize all addresses other than those specified in *addr-spec* (a logical NOT function).
PAGE	specifies a range of addresses which is exactly 256 bytes in length. A page is specified with a single 8-bit page address. Valid page addresses range from 00H to 0FFH with PAGE 00H beginning at location 0000H.
addr-spec	is a memory location or range of locations.
FROM	specifies an address where trace is to begin.
TIL	specifies an address where trace is to stop.

Discussion

Use a trace register (TRCREG) to store trace collection specifications used in the GO TRACE command. Defined trace registers (TRCREGs) are activated with the TRACE keyword and the name of the register in the GO command. Unlike breakpoints, tracepoints do not halt program execution.

Set tracepoints on events such as reaching a specific program line or label.

Manipulating TRCREGs

You manipulate a named TRCREG in the following ways:

- Create TRCREGs with the DEFINE command.
- Delete TRCREGs from memory with the REMOVE command.
- List TRCREG names with the DIR command.
- Save TRCREGs on file with the PUT/APPEND commands.
- Restore TRCREGs from files with the INCLUDE command.
- Display TRCREGs with their keywords and names.
- Execute TRCREGs with the GO TRACE command.
- Modify a TRCREG with the editor.

Examples

1. Define a TRCREG that turns on the trace option only when the program is outside of page 25H.

```
hlt> DEFINE TRCREG optrace = OUTSIDE PAGE 25H
hlt>
```

2. Define a TRCREG that captures trace information only when the program executes a certain partition.

```
hlt> DEFINE TRCREG trc2 = FROM 2080H TO 3000H
hlt>
```

Cross-Reference

DEFINE
GO
Name
PRINT

VERIFY

Specifies whether memory writes are verified

Syntax

$$\text{VERIFY} \left[= \left\{ \begin{array}{l} \text{TRUE} \\ \text{FALSE} \\ \textit{expression} \end{array} \right\} \right]$$

Where:

VERIFY	displays the current setting.
TRUE	turns the VERIFY option on. The initial setting is TRUE.
FALSE	turns the VERIFY option off.
expression	must evaluate to false (LSB = 0) or true (LSB = 1).

Discussion

The VERIFY command is used to specify whether or not memory writes during interrogation mode are to be verified by reading back the written value.

If VERIFY is TRUE and a verification fails, an error message is reported.

ICE-5100 emulator commands that write to memory execute faster when VERIFY is FALSE.

Example

1. Display the setting for VERIFY.

```
hlt> VERIFY
TRUE
hlt>
```

2. Turn the VERIFY option off.

```
hlt> VERIFY = FALSE
hlt>
```

Cross-Reference

Pseudo-variable

VERSION

Displays software version numbers

Syntax

VERSION

Discussion

The VERSION command displays the version numbers of the host and probe software, and probe firmware. These numbers should be referenced when you contact Intel regarding this product.

Example

1. Display the software version numbers.

```
hlt> VERSION
ICE-5100/nnn Version Numbers
-----------------------
Host software    -- vx.y
Probe software   -- vx.y
Probe firmware   -- vx.y
hlt>
```

WAIT

Suspends command processing by the ICE-5100 emulator until emulation is finished

Syntax

WAIT

Duscussion

The WAIT command suspends all command processing by the ICE-5100 emulator until the probe exits emulation. This feature is necessary within debug procedures, for example, to delay execution of an instruction following a GO command until emulation halts.

Example

1. The following example illustrates the use of WAIT within a PROC.

```
hlt> DEFINE PROC test1 = DO
.hlt> REPEAT
..hlt> GO TIL :int1.serv
..hlt> WAIT                        /* Cannot read DPTR in emu> mode */
..hlt> UNTIL DPTR == 0C0DH
..hlt> ENDREPEAT
.hlt> END
hlt>
```

Cross-Reference

GO
PROC

WORD

Displays or changes memory
as a 16-bit value

Syntax

```
                            ┌    ┌                              ┐ ┌       ┐ ┐
WORD [mspace] addr-spec     │ = ┤ expression                    ├ │ , . . . │ │
                            │    │ mtype [mspace] addr-spec     │ └       ┘ │
                            └    └                              ┘           ┘
```

Where:

WORD	is the keyword to display or write to a memory location.
mspace	is IDATA, XDATA, RDATA, or CODE.
addr-spec	is a memory address or a range of memory addresses.
expression	must convert to a 16-bit unsigned value.
mtype	is ADDRESS, BIT, BYTE, CHAR, or WORD.

Discussion

Use WORD to read or write the contents of memory. There are three operations that WORD can perform: read from memory, write to memory, and copy from one memory location to another. WORD is a memory object type (refer to the Mtype entry in this chapter), so it can also be used to define debug variables. WORD has the same function as ADDRESS.

Examples

1. Set a single value in XDATA, then display the value.

```
hlt> WORD XDATA 0083H = 10FA
hlt> WORD XDATA 0083H
XDATA 0083H 10FA
hlt>
```

2. Copy a single value from one memory location in IDATA to another.

```
hlt> WORD 0051H = WORD 0052H
hlt>
```

3. Define a WORD debug variable in IDATA with an initial value = 10FA.

```
hlt> DEFINE WORD bench1 = 10FA
hlt>
```

WORD (continued)

Cross-Reference

Mspace
Mtype

WRITE

Displays character strings on the screen

Syntax

WRITE [SCREEN (*x,y*)] [USING('*format-item*[, . . .]')] *write-item* [, . . .]

Where:

WRITE	displays the *write-items* on the screen.
write-item	is an ASCII character string enclosed in apostrophes ('), or a numerical expression. The maximum number of *write-items* allowed is nine.
SCREEN (*x,y*)	writes the *write-items* to a specified (*x,y*) coordinate location on the display screen. After the write, the cursor returns to its previous location.
(*x,y*)	are screen coordinates. The upper left corner is location (0,0). Columns (*x*) are numbered 1t to 80t. Rows (*y*) are numbered 0 to the maximum number of rows on your screen.
USING	formats the display according to a list of one or more *format-items*.
format-item	is one of the following:

n	specifies the width of the output field. If *n* = 0 and the number is binary or hexadecimal, the length used is the normal display length of the item without padding or truncation. If the number base is decimal, 0 specifies that the output be left-justified.
[*n*]C	moves the output pointer to column *n* (1t - 80t). The next item is written from that point. The C (without *n*) moves the pointer to the next column.
[*n*]X	writes *n* spaces between items. The X (without *n*) writes one space.
H	writes the numerical item in hexadecimal.
T	writes the numerical item in decimal.
Y	writes the numerical item in binary.
.	terminates the format string (optional). If the list contains undisplayed items, they remain undisplayed.
>	terminates the format string. The carriage return and line feed are not issued following the write. If the list contains undisplayed items, they remain undisplayed.

&	terminates the format string. The write output buffer is not flushed at the end of the write. Later writes are added to the one in the buffer. If the list contains undisplayed items, they remain undisplayed.
"*text*"	inserts ASCII text in the format. In this context only, you must enclose "*text*" in double quote marks (").

NOTE

If a BASE specifier (H, T, Y) is used, then it must precede the field width specifier (*n*) for the corresponding *write-item*.

Discussion

The WRITE command is most often used in procedures to add explanatory text to returned values or to write returned values in a more useful form, such as a table.

The WRITE command displays a maximum of 200 bytes of data. An ASCII character is one byte of data. Spaces, carriage returns, and line feeds count as characters. The byte content of numerical expressions depends on the memory type required to store the number. If you try to write more than 80 bytes of data on a line, an exclamation point (!) is displayed at the end of the line and you cannot see the rest of the information.

Unless specified in the format string by the continuation symbol (&), the information in the write buffer is deleted at the end of every write. Even if you specify the continuation symbol, the write buffer is deleted on a return from a procedure.

If the *write-list* contains more items than are specified by the format string, the format string is reused from the beginning, until all *write-items* are displayed according to the format.

Examples

1. Write a character string to the display screen.

```
hlt> WRITE 'TMO req at line 14 = ';TMO
TMO req at line 14 = FDF9H
hlt>
```

2. Format a display.

```
hlt> DEFINE BYTE aa = 45
hlt> DEFINE BYTE bb = 67
hlt> DEFINE BYTE cc = 22
hlt> WRITE USING ('T,2,2x') aa,bb,cc
45 67 22
hlt>
```

3. The following example shows the WRITE USING option. The procedure SQUAREIT squares a number the user specifies at the time the procedure is called (%0).

   ```
   hlt> DEFINE PROC squareit = DO
   .hlt> WRITE USING (' "The square of" X,T,0,X,"is",X,T,0') %0, %0*%0
   .hlt> END
   hlt>
   ```

   Calling the procedure and specifying the number:

   ```
   hlt> squareit(7)
   The square of 7 is 49
   hlt>
   ```

4. The following procedure, SQR, squares a series of numbers from 0 to a number (%0) the user specifies at the time the procedure is called. The display format is set up as a table with headers.

   ```
   hlt> DEFINE PROC sqr = DO
   .hlt> WRITE USING (' "Number" 20C,"Square" ')
   .hlt> DEFINE BYTE bb = 0
   .hlt> REPEAT
   ..hlt> UNTIL bb == %0 + 1
   ..hlt> WRITE USING ('2X,T,2,22C,T,3') bb,bb * bb
   ..hlt> bb = bb + 1
   ..hlt> END
   .hlt> END
   hlt> sqr(5)
   Number                  Square
      0                       0
      1                       1
      2                       4
      3                       9
      4                      16
      5                      25
   hlt>
   ```

Cross-Reference

Expression

Appendixes

166267-001

A STATE OF THE ICE™-5100 EMULATOR ON POWER-UP

The following list describes the state of the ICE-5100 emulator when power is first turned on.

- **Number base** is decimal. Use the BASE command to change to hexadecimal or binary.
- **Memory** is mapped to USER except for the on-chip ROM in the emulation processor (EA not asserted). Use the MAP command to map memory to either ICE or USER.
- **Symbolic debugging** is enabled. You can disassemble memory and display trace data showing symbolic references. Turn the option off by setting SYMBOLIC to FALSE.
- **Error messages** are displayed. When an emulator error is encountered, an error message is displayed. Turn the option off by setting ERROR to FALSE.
- **Temperature checking** is enabled. The user probe is polled to ensure that the controller pod's temperature does not exceed 113° F (45°C). Turn the option off by setting TEMPCHECK to FALSE.
- **External data memory writes** are verified (a read is done after every memory write). Turn the option off by setting VERIFY to FALSE.
- **Program reference pointer** (NAMESCOPE) is set to 0FFFFH. To change the namescope, set NAMESCOPE = addr-spec (addr-spec can be symbolic or numeric) after loading your program.
- **DYNASCOPE** is FALSE (NAMESCOPE is static). Setting DYNASCOPE to TRUE allows NAMESCOPE to be set to the current execution point when the execution point changes or you load another program.
- **Print clips display** is turned off. Execute a PRINT CLIPS command to enable this feature.
- **Multi-ICE emulation** is not enabled. Set SYNCSTART to TRUE to turn this option on.
- **Syntax menu** is turned on. Set MENU to FALSE to turn the menu off.
- **Default for program execution** is GO FOREVER. Use the syntax menu to learn how to change command options.
- **Tracing of emulation** is on. To selectively trace a portion of your program's execution, use the TRACE option with the GO command (e.g., GO FOREVER TRACE 15H TO 57H).
- **Breakpoints** are disabled. Execute a GO USING or a GO TIL command to enable a particular breakpoint condition.
- **Register settings.** Refer to your microcontroller handbook for a list of reset values for the microcontroller upon which your user probe is based.

- **Software invocation defaults.** The emulator assumes the following defaults when invoking the emulator software. Refer to your user probe supplement for a list of default states for your user probe.

- **BAUD rate** is 9600 for IBM hosts and 19200 for Series IV and Series III hosts. Intel recommends you try running your IBM at 19200 (specify the baud rate during software invocation, i.e., ICE*nnn* BAUD(19200)). If you experience communication problems with the IBM host, execute a RESET ICE BAUD (9600) command to change back to the default.

- **VSTBUFFER** is set to 5 KB. Change the buffer size during invocation by specifying the VSTBUFFER (*n*) option — where *n* is a number from 5 to 61 KB. Increasing the size of the VSTBUFFER reduces symbol access time.

- **CHANNEL** is 1. Software assumes emulator is connected to serial channel 1. Refer to the appropriate appendix in your installation supplement for information on selecting a nondefault serial channel.

B DEBUGGING TIPS

This appendix contains a list of miscellaneous topics you should be aware of when debugging.

<ESC>	Pressing the <ESC> key invokes the editor and retrieves the previous command from the history buffer for editing. <ESC> does not abort command entry (use <CNTL>C or <CNTL><BREAK>) nor does it halt emulation (use HALT command).
<CNTL>C	<CNTL>C does not halt emulation. <CNTL>C aborts command entry and also aborts the execution of a PROC. Note: If you are using a BRKREG which calls a PROC, enter <CNTL>C to abort the PROC (thus halting emulation). You may have to press <CNTL>C more than once to break from the PROC.
<CNTL><BREAK>	<CNTL><BREAK> serves the same function as <CNTL>C on IBM hosts.
<CNTL>D	If you are using a Series III host, <CNTL>D invokes the DEBUG-86 debug monitor. To return to the ICE-5100 emulator debug environment, press G <RETURN>.
<CNTL>E	<CNTL>E re-executes the last command entered. Use the history buffer keys to retrieve previously entered commands.
Temporary Work Device	If you are using a Series IV host, you must assign a temporary work device in order to edit trace information or load a program. Assign :WORK: to the directory containing the ICE-5100 emulator software (see Chapter 3 in the *ICE™-5100 Emulator Installation Supplement*, order number 167095, for additional information).
EXIT	The EXIT command does not halt emulation. Entering EXIT while the emulator is in emulation (emu> prompt) returns control to the host operating system. The emulator remains in emulation as long as power is applied or until you reinvoke the emulator software.
NAMESCOPE	The NAMESCOPE pseudo-variable is set to the prologue of the main module of your program. It is used as a reference point to determine the symbolic qualification needed to access program variables. NAMESCOPE does not change during program

execution. However, setting DYNASCOPE to TRUE causes NAMESCOPE to always be the same as the current execution point ($).

Break Messages

A break message is displayed when emulation halts because of a user-defined breakpoint. The display lists the next executable address, not the last address executed.

Trace Display

The instruction disassembly for the trace display is not done in real-time. The data is taken from memory and reconstructed for purposes of display once emulation has halted.

Changing the excution point with a $ or GO FROM command causes trace collection to begin with an empty trace buffer. Use the EDIT TRACE command to edit and save the contents of the trace buffer if desired.

Breakpoint and Tracepoint Removal

Removing a BRKREG or TRCREG definition from memory does not remove the hardware breakpoints and tracepoints set by the GO command. Execute a GO FOREVER TRACE or a GO USING *new-break-spec* TRACE command to reset the hardware.

Editing Trace Data

Executing a Q(uit) E(xecute) command while editing trace data causes the ICE-5100 emulator to try and execute the contents of the trace buffer. Exit from the editor by typing Q(uit) A(bort).

Assembly/Disassembly of Code

The ASM command assembles and disassembles program code. On previous Intel emulators, there was a DASM command for disassembling.

Disassembly of program code may be slow if a large number of symbols was loaded with your program. The following options are available to increase the speed of disassembly:

- Increase the resident symbol table size by using the VSTB option during software invocation (see ICE*nnn* entry in Chapter 3 for details).
- Set SYMBOLIC to FALSE to disable symbolic support.

Symbolic Display of Program Variables

The following restrictions apply to the symbolic display of program variables during ASM and PRINT commands:

- Addresses in the special function register space are displayed according to their reserved SFR keywords (regardless of any program declarations). Refer to your user probe supplement for a list of SFRs.
- When two or more program symbols are located at the same address (e.g., overlayed procedure variables), the symbol name displayed is the one that occurs first in the loaded object file.

ASM-51 Program Support

If an ASM-51 program was loaded with discontinuous code segments within a module, NAMESCOPE will not work properly. You will need to fully-qualify symbolic references.

The ICE-5100 emulator does not recognize symbols when using the ASM-51 constructs EQU and SET in the following format (expression is a number):

symbol [EQU] [SET] *expression*

PL/M-51 Program Support

The last line number of a PL/M module is not displayed during disassembly nor does LSTEP stop at it.

NAMESCOPE does not work properly if the NODEBUGLINES option was used when linking your program with RL51.

When using based variables, the ICE-5100 emulator only recognizes variables used as a base and not the based variable. For example, refer to the code segment below:

```
DOS 3.10 (033-P) PL/M-51 V1.2
COMPILER INVOKED BY: C:\DOSLANG\PLM51.EXE BASED.PLM DEBUG
1      1  plm_main: do;
2      1  declare bs$ptr word;
       1  declare bsvar based bs$ptr byte;
4      1  bs$ptr = 22;
5      1  bsvar = 99;
6      1  end;
```

After loading the program, bs$ptr will be usable as a variable but bsvar will not. References to bsvar can be done by finding the value in bs$ptr and using that to indirectly reference bsvar. For example:

```
hlt>WORD.bs$ptr
IDATA 0019 0022HH
hlt>BYTE 22H
99
hlt>
```

You can also reference bsvar by defining a LITERALLY definition. For example:

```
hlt>DEFINE LITERALLY bsvar = 'BYTE (WORD.bs$ptr)'
hlt>
```

Loading Additional Programs Into Memory

If you load another program into memory for execution, the program counter does not get reset. Execute a GO FROM 0 or $ command to reset the program counter.

Power-Down Mode

If your emulation processor is in power-down mode (bit 2 of the PCON register set), the ICE-5100 emulator will not halt emulation (via the HALT command). Executing a RESET ICE or

RESET CHIP command does not restore processor operation. You must assert a user reset of the target system to properly exit power-down mode. Alternatively, momentarily remove the V_{CC} source from the emulation processor, then execute a RESET CHIP command. If using the ICE-5100 emulator in stand-alone mode, momentarily turn the power off to the emulator, then execute a RESET ICE command.

Trouble-Shooting Tips

The following list describes some commonly encountered hardware error messages. The errors are easily corrected in most instances by becoming aware of the problem causes. Possible causes are listed after the error messages.

277	Probe Interface error.
391	Processor module is not responding.
309 and 392	Probe hardware malfunction.
393	Probe hardware malfunction. Processor was reset.
403	Internal error.
404	Bond-out device is not responding.
408	Probe is not ready.

These errors are usually caused by one of the following problems:

- V_{CC} not at emulation processor — Emulation processor not powered for a variety of reasons (e.g., target system not powered up or it has a circuit fault, circuitry incorrectly wired, etc.).
- The fuse in socket F1 on the underside of the processor module is blown. Refer to the *ICE™-5100 Emulator Installation Supplement*, order number 167095, for information on changing the fuse.
- Clock not present at emulation processor — Emulation processor clock is not present for a variety of reasons (no crystal, no external clock, circuitry incorrectly wired, etc.).

Jumpers may be set incorrectly on the bottom of the processor module. Refer to your user probe supplement for information on changing the jumper positions. NOTE: Not all user probes may have jumpers.

- User reset is asserted — Halting emulation with user reset asserted will cause a hardware error message. Check the reset pin in the target system.
- Probe software load error — When invoking the software or executing a RESET ICE command, the probe software was not loaded successfully. Ensure that the file ICE*nnn.nnn* is in the same *pathname* as ICE*nnn*.86 (or ICE*nnn*.EXE).

- Emulation processor is in power-down mode — If your emulation processor is in power-down mode (bit 2 of the PCON register set), the ICE-5100 emulator will not halt emulation (via the HALT command). Executing a RESET ICE or RESET CHIP command does not restore processor operation. You must assert a user reset of the target system to properly exit power-down mode. Alternatively, momentarily remove the V_{CC} source from the emulation processor, then execute a RESET CHIP command. If using the ICE-5100 emulator in stand-alone mode, momentarily turn the power off to the emulator, then execute a RESET ICE command.
- Serial cable disconnected — Ensure that the cable is firmly connected to the host and the ICE-5100 controller pod. Execute a RESET ICE command to reinitialize the emulator.

C USING THE CLIPS ASSEMBLY

C.1 Introduction

The ICE-5100 emulator's clips assembly provides eight bits of external data as well as the trace qualification signals $\overline{\text{TQIN}}$ and $\overline{\text{TQOUT}}$. External trace data can be optionally latched with the $\overline{\text{HOLD}}$ line. Two ground clips are available with the other clips lines.

The following sections describe using the clips assembly.

C.2 Connecting the Clips Assembly

Perform the following steps to connect the clips assembly to the controller pod and the target signal lines.

1. Connect the clips assembly to the ICE-5100 controller pod as described in the *ICE™-5100 Emulator Installation Supplement*, order number 167095.

2. Connect one or both ground clips to the common or ground line of the circuit to be measured. This should be at the same potential as the V_{SS} pin on the target microcontroller socket, if present.

3. Connect the eight data clips to the selected signals to be measured. The signals should not exceed +7V with respect to the ground clips.

C.3 Clips Operation

If $\overline{\text{HOLD}}$ is not used, data is acquired with approximately the same timing as an external code fetch with relation to PSEN (refer to Table C-1 for timing specifications). $\overline{\text{HOLD}}$ can be used to latch data at some other point in the machine cycle, such as address collection with relation to ALE. Data is latched in on the falling edge of the signal connected to $\overline{\text{HOLD}}$. ALEE and PSENE are clock signals available at the P1 connector on the end of the processor module. They are similar to ALE and $\overline{\text{PSEN}}$ except they are always present. (Refer to the *ICE™-5100 Installation Supplement*, order number 167095, for information on the P1 connector.)

Clips data is collected during emulation or with the ISTEP or LSTEP commands.

C.4 External Trace Qualification Lines

The following sections discuss the trace qualification lines, TQIN and TQOUT.

C.4.1 $\overline{\text{TQIN}}$

After attaching the ground clips, the $\overline{\text{TQIN}}$ (trace qualification in) signal line can be connected to external circuitry (not to exceed + 7V with respect to the ground clips). Executed addresses and clips data are collected in the trace buffer during emulation if $\overline{\text{TQIN}}$ is active low and a valid TRACE option is specified with the GO command. For example:

```
hlt> GO TRACE 0FFF0H   /* 0FFF0H is outside of the program; */
                                           /* enable TQIN */
emu>
```

When emulation is entered, tracing occurs at address FFF0H or whenever $\overline{\text{TQIN}}$ is at a logic low. Various combinations of trace qualification can be used with the GO command, but at least one trace address has to be specified. This address is usually outside the expected range of emulation. The $\overline{\text{TQIN}}$ line is initially disabled as TRACE is qualified for all addresses. It is also disabled when executing the folowing emulation trace command:

```
    hlt> GO TRACE       /* Trace all addresses; disable TQIN */
    emu>
```

C.4.2 $\overline{\text{TQOUT}}$

The $\overline{\text{TQOUT}}$ (trace qualification out) signal is a TTL level output that is active low when trace is qualified. $\overline{\text{TQOUT}}$ is useful for timing program execution through certain address ranges or gating external test equipment such as logic analyzers.

The initial default trace qualification is to trace all execution addresses during emulation. In this case, $\overline{\text{TQOUT}}$ is low during emulation. If a specific trace qualification is used then $\overline{\text{TQOUT}}$ is low only when emulation is occurring in the specified address range.

The following example causes $\overline{\text{TQOUT}}$ to be low only during emulation in the address range 0 to 100H.

```
    hlt> GO TRACE 0 to 100H
    emu>
```

C.5 Displaying Clips Data

Use the PRINT CLIPS command to display the clips data in binary. The clips data is displayed to the right of the trace data. The following example enables the display of clips data whenever you display trace data:

```
    hlt> PRINT CLIPS
```

PRINT CLIPS remains enabled until you execute a PRINT NOCLIPS command (initial condition).

For more information, refer to the PRINT entry in Chapter 3.

C.6 AC and DC Specifications for Clips Assembly

Table C-1 lists the AC specifications for the emulation clips assembly.

Table C-1 AC Specifications for the Clips Assembly

Signal	MIN	MAX
Data 0-7, TQIN input capacitance		15-20 pf
Data 0-7, set-up time	55 ns	
Data 0-7, Hold time		0 ns
$\overline{\text{HOLD}}$ data set-up time	10 ns	
$\overline{\text{HOLD}}$ data hold time	3 ns	
$\overline{\text{HOLD}}$ pulse width	6 ns	

Table C-2 lists the DC specifications of the emulation clips assembly.

Table C-2 DC Specifications for the Clips Assembly

DC Specifications	MIN	MAX
Data 0-7		
Input voltage, V_i	−0.5 V	7.0 V
Input current, I_{ih}		100 μA
Input current, I_{il}		−.6 mA
Positive threshold, V_{t+}	1.4 V	2.0 V
Negative threshold, V_{t-}	0.5 V	1.1 V
Input hysterisis	0.4 V	
$\overline{\text{TQIN}}$, $\overline{\text{HOLD}}$		
Input current, I_{il}		−1.1 mA
$\overline{\text{TQOUT}}$		
Output current, I_{oh}		−1.0 mA
Output current, I_{oh}		20 mA

D HARDWARE SPECIFICATIONS

This appendix contains specifications on the power supply, serial cable pin-outs, and clips assembly.

D.1 Power Supply

Table D-1 lists the power supply specifications.

Table D-1 Power Supply Specifications

Power Supply Characteristics	Specifications
AC disconnect	Manual switch
Mount	Freestanding
Cooling	Convection
Dimensions	7in x 4in x 11in 18 cm x 10 cm x 28 cm
Weight	14 lbs (6.1 kg)
AC protection	Fuse
Peak DC output current	10A
Maximum continuous DC output current @ 5V	8A
Minimum continuous DC output current @ 5V	.08A
Maximum short circuit current @ 5V	6A
Minimum efficiency @ 5V	30%
Ripple @ 5V	50mV
Holdup time	10ms
Peak output current @ ± 12V	1A
Maximum continuous DC output current @ ± 12V	75ma
Ripple @ ± 12V	100mV
AC line frequency variation	± 3Hz
AC line voltage fluctuation	± 10%
Primary voltage selections	100V 120V 220V 240V
Primary voltage selection	Switch
Overvoltage protection threshold	6.45V ± .45V
DC regulation	± 2%
Frequency	50/60 Hz

D.2 Serial Cable

Table D-2 lists the serial-cable pin-out signals at the controller pod. Figure D-1 shows the signal connections for the serial cable and the controller pod.

Table D-2 Serial Cable Pin-Outs At Controller Pod

Pin	Signal	RS-232 Name	Direction
1	Carrier detect (CD)	CF	Send
2	Received data (RXD)	BB	Send
3	Transmitted data (TXD)	BA	Receive
4	Data terminal ready (DTR)	CD	Receive
5	Signal ground	AB	—
6	Data set ready (DSR)	CC	Send
7	Request to send (RTS)	CA	Receive
8	Clear to send (CTS)	CB	Send
9	Ring indicator	CE	Send

Table D-3 lists the serial-cable pin-outs for the host connectors.

Table D-3 Serial Cable Pin-outs at Host Conector

Signal	Pins on Host Connector		
	Series III/IV	PC AT	PC XT
Chassis ground	1	-	1
Received data (RXD)	2	2	3
Transmitted data (TXD)	3	3	2
Request to send (RTS)	4	7	4
Clear to send (CTS)	5	8	5
Data set ready (DSR)	6	6	6
Signal ground	7	5	7
Carrier detect (CD)	8	1	8
Data terminal ready (DTR)	20	4	20
Ring indicator	22	9	22

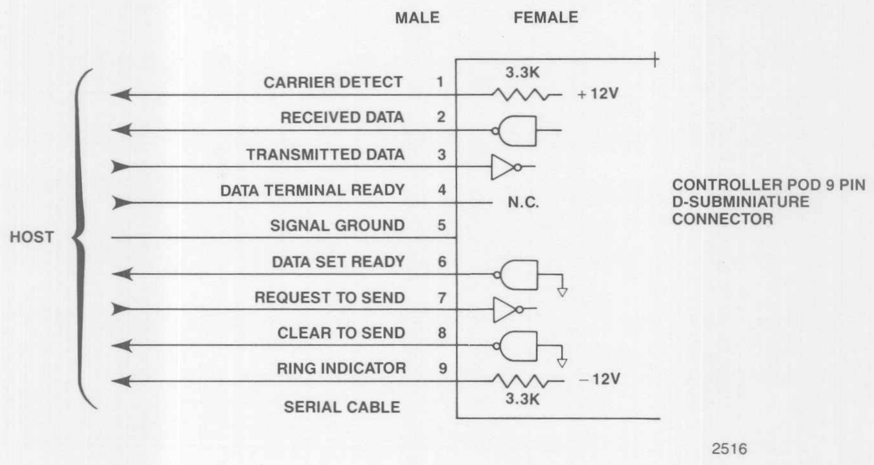


Figure D-1 Serial Cable Signal Connections

E ERROR MESSAGES

The five classes of errors that the ICE-5100 emulator reports are as follows:

WARNING	The ICE-5100 emulator takes no action and command processing continues. Warnings advise you of a possible error condition.
ERROR	The ICE-5100 emulator stops processing the current command. The prompt reappears, indicating that you should try the command again.
SEVERE ERROR	The ICE-5100 emulator closes all INCLUDE files and the prompt reappears. Memory may be altered.
FATAL ERROR	Non-recoverable error has occurred. Control will return to the host operating emulator. Memory may be altered.
INTERNAL ERROR	Indicates an internal software problem. The ICE-5100 emulator stops processing the current command and the prompt reappears. Memory may be altered. These errors should never occur and should be reported to Intel if encountered. Refer to the inside back cover for service information.

All error mesages have the following format:

severity-level #number
message[*]

Where:

severity-level	is WARNING, ERROR, SEVERE ERROR, FATAL ERROR, or INTERNAL ERROR.
#number	is the error message number.
message	is the text of the error. If the ERROR command is set to FALSE, the error message display is suppressed.

Messages followed by a [*] have additional information. The extended message is displayed on-line by entering HELP E*n*, where *n* is the number of the error message. For example, to obtain the extended message for error #9, enter:

```
hlt>HELP E9
```

List of Error Messages

The following is a list of error messages displayed by the ICE-5100 emulator. Most of the error messages are self-explanatory; if the error message has an extended error message (indicated by [*]), it is listed below the error message.

Some error messages contain references to other parts of this manual where additional information can be obtained on how to rectify and avoid making the error.

0 Type definition record with unrecognizable format.

1 Address of module is not known.

6 Unknown module specified.

7 No line information was loaded for module.

Refer to the LOAD command in Chapter 3 for information on loading symbolic information with your program.

8 No symbol information was loaded for module.

Refer to the LOAD command in Chapter 3 for information on loading symbolic information with your program.

9 Cannot determine module for specified location. [*]

*Could not find specified location in any known module. The specified location is either outside of the program or in a module for which there is no symbol information.

10 Cannot determine current default module. [*]

*Could not find current location in any known module. The current execution point is either outside of the program or in a module for which there is no symbol information.

12 Symbol not known in current context. Symbol is either not known, or is not local to the current module or procedure.

Refer to the DYNASCOPE and NAMESCOPE commands is Chapter 3 for information on creating program references.

13 No symbol information is loaded for the program.

14 Attempt to reference a program symbol of an unsupported type.

20 Specified line is not an executable statement.

21 Specified line does not exist in module.

22 Cannot evaluate line reference. [*]

*No symbol information was loaded for the module.

Refer to the LOAD command in Chapter 3 for information on loading symbolic information with your program.

23 Specified type is incompatible with directory. [*]

*Specified type cannot be used with the specified (or default) directory. For example, DIR PUBLICS LINE is contradictory.

Refer to the DIR command in Chapter 3 for information on using the DIR command to display information.

24 Cannot perform symbol table request. No user program loaded.

40 Tried to REMOVE debug object declared locally in DO..END block.

Refer to the DEFINE command in Chapter 3 for information on defining debug objects.

41 Workspace exceeded. [*]

*Out of workspace. Delete any unnecessary debug objects (e.g., PROCs, LITERALLYs). This could also be caused by deeply recursive PROCs.

If you receive this error often, it may indicate a need for more host memory. Refer to the REMOVE command in Chapter 3 for information on removing debug objects from memory.

42 The name is either undefined or not of the correct type.

43 The name is undefined.

Refer to the DEFINE command in Chapter 3 for information on creating debug objects.

44 The name is already defined with a different type.

Refer to the DEFINE command in Chapter 3 for information on creating debug objects.

45 Parameter is outside the body of a PROC.

Refer to the PROC command in Chapter 3 for information on creating and using debug procedures.

46 The name is not a PROC.

Refer to the DEFINE and DIR commands in Chapter 3 for information on defining and accessing debug procedures.

47 Illegal type specified in DIR DEBUG command.

Refer to the DIR command in chapter 3 for information on using the DIR command to display debug definitions.

48	The named object is not a LITERALLY. Refer to the DIR command in Chapter 3 for information on displaying LITERALLYs.
49	Illegal assignment to a register. Refer to the BRKREG and TRCREG commands in Chapter 3 for information on debug registers.
51	Error in debug symbol lookup. [*] *May be caused by removing a global object referenced in a local context (e.g., a PROC), and then executing that reference (calling the PROC).
52	Type illegal for PUT or APPEND. Refer to the APPEND and PUT commands in Chapter 3 for information on which types of debug objects can be saved to a disk file.
53	Type illegal for DEFINE. Refer to the DEFINE command in Chapter 3 for the syntax of the DEFINE command.
54	Type illegal for REMOVE. Refer to the REMOVE command in Chapter 3 for information on removing debug object definitions from memory.
64	Attempt to PUT or APPEND a local debug object.
65	I/O error on PUT file.
66	This command is not currently implemented.
67	This command is not allowed inside of a compound command.
69	Invalid type conversion. Refer to the Expressions topic in Chapter 3 for information on type conversions.
70	String longer than 254 characters.
71	String too long for numeric conversion. In order to convert character strings to unsigned numbers, they must be of length 0 or 1.
74	WRITE list too long. Refer to the WRITE command in Chapter 3 for the proper syntax.
75	WRITE data too large. Refer to the WRITE command in Chapter 3 for the proper syntax.
76	Invalid format string in WRITE command. Refer to the WRITE command in Chapter 3 for the proper syntax.

77 Address or partition does not exist in <mspace>. [*]

*The address or partition must be entirely within the specified memory space. For example, the partition 60K LENGTH 5K is an invalid reference to the CODE <mspace> because program memory contains only 64K bytes.

79 Invalid expression for MTYPE.

80 Invalid Boolean operation.

Refer to the Expressions topic in Chapter 3 for information on Boolean operations.

81 Invalid string operation.

Refer to the Expressions topic in Chapter 3 for information on string operations.

82 Memory space conflict in operation. [*]

*This could be caused by performing a Boolean operation on two program symbols which exist in different <mspace>s. The expression (.var1 < .var2) comparing the addresses of var1 and var2 would be illegal if var1 is in XDATA and var2 is in IDATA.

83 Invalid operation.

Refer to the Expressions topic in Chapter 3 for information on valid operations.

84 Attempt to assign a value to a symbolic location in CODE memory. [*]

*An attempt was made to assign an expression to a location associated with CODE memory (e.g., :main.proc1 = 5, where proc1 is a procedure in module main). Straight assignments can only be made to variables or data located in data memory (e.g., byte xdata 80 = 5).

85 Cannot use editor if ICE-5100 was invoked with SUBMIT control.

88 ICE-5100 stack overflow — probably due to deep recursion of a PROC.

90 LITERALLY nesting too deep.

This error usually occurs when one literal uses another literal as part of its definition.

91 Invalid bit operation.

Refer to the Expressions topic in Chapter 3 for information on valid bit operations.

94 The number #<n> specified in HELP E<n> must be a decimal number.

95 Invalid partition. [*]

*A memory partition was specified in which the first address is greater than the second address. For example, ASM 200H TO 100H is an invalid partition.

Refer to the Address topic in Chapter 3 for information on specifying a valid memory partition.

96 Too many source items were specified within a memory modify command.

98 Command cannot be performed during emulation.

Refer to the GO command in Chapter 3 for information on commands which can not be entered during emulation.

99 The specified break conditions exceed break memory resources. [*]

*The break hardware will support a maximum of four individual breakpoints. A range breakpoint or an OUTSIDE breakpoint is equivalent to three individual breakpoints and any number of page breakpoints are equivalent to one individual breakpoint.

100 An invalid baud rate value was specified with the BAUD option.

Refer to the RESET command in Chapter 3 for information on resetting the emulator software using a valid baud rate.

101 An invalid channel value was specified with the CHANNEL option.

Refer to the RESET command in Chapter 3 for information on resetting the emulator software using a valid serial channel.

102 FIFO is not in freeze mode.

103 Cannot write to CODE memory which is mapped to USER.

Refer to the MAP command in Chapter 3 for information on mapping memory.

106 Load file contains relocatable symbol(s). [*]

*Possibly because the object file was not generated by RL51 and, therfore, contains non-absolute symbol definitions. The records containing non-absolute symbols are ignored during the load.

108 Load file contains invalid absolute OMF record(s). [*]

*Possibly because the object file was not generated by RL51 and, therefore, contains non-absolute OMF or fixup records. These types of records are ignored during the load.

109 Load file is not a valid MCS-51 object file.

Refer to the LOAD command in Chapter 3 for information on the types of files which can be loaded for emualtion.

115 Bad object record in the load file.

119 Memory segment request failure during operation. [*]

*More memory is needed to perform operation. Deletion of debug objects will not increase available memory.

136 Divide by zero (operation yields 0 result).

Refer to the Expression topic in Chapter 3 for valid arithmetic operations.

137 Invalid type for arithmetic.

Refer to the Expression topic in Chapter 3 for valid arithmetic operations.

138 Invalid integer operation.

Refer to the Expression topic in Chapter 3 for valid arithmetic operations.

144 Illegal numeric constant.

Refer to the Expression topic in Chapter 3 for valid arithmetic operations.

160 Attempt to INCLUDE :CI:.

161 I/O error on INCLUDE file.

162 I/O error on LIST file.

163 I/O error while loading object file.

165 Error while attempting to open the Virtual Symbol Table.

166 Error while attempting to seek in the Virtual Symbol Table.

167 Error while attempting to write to the Virtual Symbol Table.

168 Error while attempting to close the Virtual Symbol Table.

169 Error while attempting to read the Virtual Symbol Table.

170 Number to be converted to a string occupies more than 50 bytes.

171 Too much text was specified for the write buffer.

173 Format used in WRITE command was incorrect. [*]

*See HELP WRITE for information on the WRITE command.

174 The type of an element in an output list was incorrect.

Refer to the WRITE command in Chapter 3 for the proper syntax.

176 A quoted string within a WRITE format is not terminated (e.g., USING ('"hi')).

177 Invalid number for the currently specified base.

Refer to the BASE command in Chapter 3 for the correct syntax for use with the BASE command.

178 Illegal use of the up-arrow (↑).

179 Invalid string to number conversion.

Refer to the STRTONUM command in Chapter 3 for information on performing string to number conversions.

181 Illegal base value. BASE will only accept assignments to values equivalent to 2, 10, or 16 decimal.

182 Unable to create file.

183 An error occurred while writing to the file.

201 Load error. [*]

*Data in the Virtual Symbol Table was corrupted during or following a load.

248 Illegal syntax in the CRT file.

Refer to the ICE*nnn* command in Chapter 3 for information on using a CRT file.

249 Insufficient memory for the Virtual Symbol Table buffers.

Refer to the ICE*nnn* command in Chapter 3 for information on expanding the virtual symbol table.

250 Error occurred while trying to load a system overlay. [*]

*This probably occurred because the disk from which the system software was loaded is not in the same drive as it was when it was invoked.

266 Fatal disk file error.

267 Probe software load error. [*]

*An error occurred while attempting to load the probe software. This could have been because the probe software file could not be found or there was a checksum error in the probe software file.

277 Probe interface error. [*]

*Serial communication failed between the host and probe. Issue a RESET ICE command to restore serial communication.

Refer to Appendix B for a list of conditions that could cause this type of error and information on correcting the situation.

287 No break registers are defined.

Refer to the BRKREG command in Chapter 3 for information on creating BRKREGs.

353 Illegal number.

356 Illegal single line assembler operand.

357 Illegal length (null) string specified as ASM text.

359 Too few operands for this instruction.

361 The types of the operand(s) do not match the mnemonic or each other.

362 One byte relative jump is out of range. Range is − 128 to + 127.

363 Invalid immediate operand.

364 Invalid indirect operand. [*]

*Only R0, R1, DPTR, A + DPTR, and A + PC may appear after an @ sign for indirect addressing.

366 Unrecognized mnemonic used in instruction.

368 Invalid address used in instruction requiring reference to a bit address.

370 Invalid use of the slash (/). [*]

*A slash (/) followed by a bit address can only be used as the second operand in the following instructions:

MOV C,/bit-address

ANL C,/bit-address

ORL C,/bit-address

371 Invalid destination address for ACALL or AJMP.

372 Operand value is too large.

380 Invalid <sfr name> specified for the current CPU. [*]

*A special function register name was specified which does not exist for the CPU type. For example, TH2 would be an invalid <sfr name> if the CPU being emulated was an 80C51.

Refer to your user probe supplement for a list of valid special function registers that are accessible to your user probe.

381 Cannot repeat instructions, code block too large.

382 Incomplete repetition of specified instruction(s).

383 Symbolic reference is not in CODE memory. [*]

*Caused by specifying a program address with a symbol which is not defined in program memory. For example, if VAR1 is a program variable, the command GO TIL .VAR1 is invalid because .VAR1 does not refer to a CODE address.

385 Symbolic references in the ASM command must be fully qualified.

Refer to the Symbolic References entry in Chapter 3 for information on symbolic references to program variables.

389 Instruction step size was changed to match the hardware. [*]

*The specified expression for ISTEP was greater than the maximum step size of 255 (0FFH). The expression was changed to the maximum step size prior to ISTEP execution.

Refer to the ISTEP command in Chapter 3 for information on using the ISTEP command.

390 Probe hardware malfunction. Current program counter and newest trace frame may be invalid.

Refer to Appendix B for a list of conditions that could cause this type of error and information on correcting the situation.

391 Processor module is not responding.

392 Probe hardware malfunction. [*]

*Processor may be in an unknown state, and the current program counter and newest trace frame may be invalid. A RESET CHIP command is recommended.

Refer to Appendix B for a list of conditions that could cause this type of error and information on correcting the situation.

393 Probe hardware malfunction. Processor was reset.

Refer to Appendix B for a list of conditions that could cause this type of error and information on correcting the situation.

400 Memory verify error while writing. [*]

*Read after write check found the value read did not match the value written. Possible causes:

— Attempted write to ROM or nonexistent memory

— Bad RAM

The error message can be overridden by setting the VERIFY pseudo-variable to FALSE, however, the write error will still exist.

Refer to the VERIFY command in Chapter 3 for information on using the VERIFY command.

404 Bond-out device is not responding. [*]

*Possible causes:

— Probe is not receiving power

— Bond-out clock not present

— Bond-out is in Power Down mode

— Bond-out malfunction

Refer to Appendix B for a list of conditions that could cause this type of error and information on correcting the situation.

405 Invalid <bit number> specified. [*]

*Only <bit number>s 0 through 7 are valid.

406 Specified register is not bit-addressable. [*]

*See HELP REGS for information on bit-addressable registers.

407 Invalid register bank selected. [*]

*Valid register banks are 0 through 3.

Refer to the RBANK command in Chapter 3 for information on selecting a register bank.

408 Probe is not ready. [*]

*Possible causes:

— Probe software load error

— Bond-out malfunction

Refer to Appendix B for a list of conditions that could cause this type of error and information on correcting the situation.

409 ISTEP or LSTEP not allowed when SYNCSTART is enabled.

Refer to the SYNCSTART command in Chapter 3 for information on mult-ICE emulation.

514 A return type of Boolean is expected.

Refer to the PROC command in Chapter 3 for information on returning values from a debug procedure.

528 Recursion in the register definition is not allowed.

Refer to the DEFINE command in Chapter 3 for information on creating debug objects.

530 BRKREG/TRCREG undefined.

Refer to the DEFINE and DIR commands in Chapter 3 for information on creating and displaying BRKREG and TRCREG definitions.

547 Probe or serial port is not responding. [*]

*Check probe power and serial connections, then issue a RESET ICE command.

Refer to Appendix B for a list of conditions that could cause this type of error and information on correcting the situation.

F ASCII CODES

This appendix lists ASCII codes and their functions. Use the EVAL command to convert numbers from one number base to another.

Table F-1 contains a list of ASCII codes.

Table F-2 contains a list of ASCII code functions.

F.1 ASCII Tables

Table F-1 ASCII Code List

Dec	Hex	Character	Dec	Hex	Character
0	00	NUL	32	20	SP
1	01	SOH	33	21	!
2	02	STX	34	22	“
3	03	ETX	35	23	#
4	04	EOT	36	24	$
5	05	ENQ	37	25	%
6	06	ACK	38	26	&
7	07	BEL	39	27	’
8	08	BS	40	28	(
9	09	HT	41	29	)
10	0A	LF	42	2A	*
11	0B	VT	43	2B	+
12	0C	FF	44	2C	,
13	0D	CR	45	2D	—
14	0E	SO	46	2E	.
15	0F	SI	47	2F	/
16	10	DLE	48	30	0
17	11	DC1	49	31	1
18	12	DC2	50	32	2
19	13	DC3	51	33	3
20	14	DC4	52	34	4
21	15	NAK	53	35	5
22	16	SYN	54	36	6
23	17	ETB	55	37	7
24	18	CAN	56	38	8
25	19	EM	57	39	9
26	1A	SUB	58	3A	:
27	1B	ESC	59	3B	;
28	1C	FS	60	3C	<
29	1D	GS	61	3D	=
30	1E	RS	62	3E	>
31	1F	US	63	3F	?

Table F-1 ASCII Code List (continued)

Dec	Hex	Character	Dec	Hex	Character
64	40	@	96	60	\
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z
91	5B	[	173	7B	{
92	5C	\	174	7C	|
93	5D	]	175	7D	}
94	5E	^	126	7E	~
95	5F	_	127	7F	DEL

Table F-2 ASCII Code Definition

Abbreviation	Meaning	Dec	Hex
NUL	NULL Character	0	0
SOH	Start of Heading	1	1
STX	Start of Text	2	2
ETX	End of Text	3	3
EOT	End of Transmission	4	4
ENQ	Enquiry	5	5
ACK	Acknowledge	6	6
BEL	Bell	7	7
BS	Backspace	8	8
HT	Horizontal Tabulation	9	9
VT	Vertical Tabulation	11	0B
CR	Carriage Return	13	0D
SO	Shift Out	14	0E
SI	Shift In	15	0F
DLE	Data Link Escape	16	10
DC1	Device Control 1	17	11
DC2	Device Control 2	18	12
DC3	Device Control 3	19	13
DC4	Device Control 4	20	14
NAK	Negative Acknowledge	21	15
SYN	Synchronous Idle	22	16
ETB	End of Transmission Block	23	17
CAN	Cancel	24	18
EM	End of Medium	25	19
SUB	Substitute	26	1A
ESC	Escape	27	1B
FS	File Separator	28	1C
GS	Group Separator	29	1D
RS	Record Separator	30	1E
US	Unit Separator	31	1F
SP	Space	32	20
DEL	Delete	127	7F

Glossary

166267-001

G Reference Publications

The publications described in the following sections contain information you may find helpful. Copies of the publications are available through the Intel Literature Department (see page ii of this manual for the address).

G.1 Hardware Reference Publications

- *Memory Components Handbook*, order number 210830.
  This catalog contains data sheets on the memory components manufactured by the Intel Corporation.
- *Microcontroller Handbook*, order number 210918.
  This catalog contains data sheets on the microcontroller products manufactured by the Intel Corporation.
- *OEM Systems Handbook*, order number 210941.
  This catalog contains data sheets on integrated microcomputer systems, single-board computers, memory expansion boards, high-speed math boards, peripheral controllers, communications controllers, digital I/O expansion and signal conditioning boards, industrial control series, and analog I/O expansion. It also contains data sheets on systems software, such as the iRMX™ operating system, and the DCX-51 Executive.
- *Development Systems Handbook*, order number 210940.
  This catalog contains data sheets on microcomputer development systems (hardware and software), in-circuit emulators, network development systems, system design kits, and third-party software.

G.2 Software Reference Publications

- *MCS®-51 Macro Assembler User's Guide*, order number 9800937
  This manual provides design and operating information about the ASM-51 assembly language. It also provides an overview of the language, procedures for program structuring, information about data operation, and an appendix describing the 8051 instruction set.
- *MCS®-51 Utilities User's Guide*, order number 121737
  This manual describes the RL51 linker and relocator utilities.
- *PL/M-51 User's Guide*, order number 121966.
  This manual provides programming instructions for PL/M-51. It includes details on expressions and assignments, procedures, variables, and a sample program.

- *MCS®-51 Macro Assembler User's Guide for DOS*, order number 122752.
  This manual provides design and operating information about using ASM-51 assembly language with DOS systems. It also provides an overview of the language, procedures for program structuring, information about data operation, and an appendix describing the 8051 instruction set.
- *MCS®-51 Utilities User's Guide for DOS*, order number 122747
  This manual describes the RL51 relocator and linker process for use with DOS systems.
- *PL/M-51 User's Guide for DOS*, order number 122742
  This manual provides programming instructions for using PL/M-51 on DOS systems. It includes details on expressions and assignments, procedures, variables, and a sample program.

GLOSSARY

Asserted: A signal driven to its true state. See Released.

Bond-out device: Another name for the emulation processor. See Emulation processor.

Breakpoint: Execution address where program execution is to halt.

Clips: Eight bits of trace available to the user as clips leads.

Control processor: The microcontroller in the controller pod used to control the communications with the host for all ICE functions.

Controller pod: A chassis containing a single PBA and electronics to control the emulation processor.

CPA: Crystal power accessory. Used to attach the target processor to the ICE-5100 emulator and to provide power and clock to the processor module. Also implements port pin diagnostics.

Emulation: Activity transpires identical to the target processor.

Emulation memory: 64 KB of memory. Accessible to the control procesor for reading and writing only while in ICE mode. Accessible to the target processor for code fetches in emulation. Intended for use as target processor's ROM.

Emulation processor: Microcontroller used to emulate the target processor. Contains additional circuits to implement an ICE interface.

ESD: Electrostatic discharge.

Execution: Events associated with instruction stream processing.

External test connector: Enables you to connect test equipment to the ICE 5100 emulator. It is located on the processor module.

Host: Computer that controls the ICE-5100 emulator through a serial link.

ICE: In-circuit emulator.

Halt mode: The emulation processor is not emulating and the I/O functions of the chip are halted.

Mspace: Acronym for memory space. Refers to one of the 8051 microcontroller's four distinct memory address spaces (e.g., CODE).

Mtype: Acronym for memory type. Refers to the format in which information is stored in computer memory (e.g., BYTE).

Multi-ICE connector: Enables you to connect multiple Intel emulators to the ICE-5100 emulator for group emulation. It is located on the controller pod.

Pathname: Is the fully-qualified reference to a file. May include device, directory, and file name.

PBA: Printed board assembly. Sometimes refered to as PB (printed boards).

PGA: Pin grid array package used for high leadcount devices.

Processor module: Houses the emulation processor, buffer ICs, and the external test connector.

Pseudo-variable: Is a cross between a command and a variable. See the Pseudo-variable entry in Chapter 3 of this manual.

Released: A signal driven to its false state. See Asserted.

SBX: 36-pin connectors used to connect the processor module to the target adaptor.

SFR: Special function register.

Serial link: Serial communications link between the ICE-5100 system and the host.

SYNC 0: Synchronization line to coordinate multi-ICE emulation start and break.

Target adaptor: Package adaptor that mounts to the bottom of the processor module and provides pin-out compatibility with the target prototype.

Target system: Electronics within which a real microcontroller (target processor) will eventually reside.

Trace memory: 254 24-bit frames of trace memory used to trace execution activity of the target processor during emulation.

Tracepoint: The execution address at which the trace qualifier will be enabled or disabled.

User cable: The flexible circuit cable that connects the controller pod to the processor cable.

User probe assembly: Assembly consisting of the user cable, processor module, and target adaptor.

Index

166267-001

INDEX